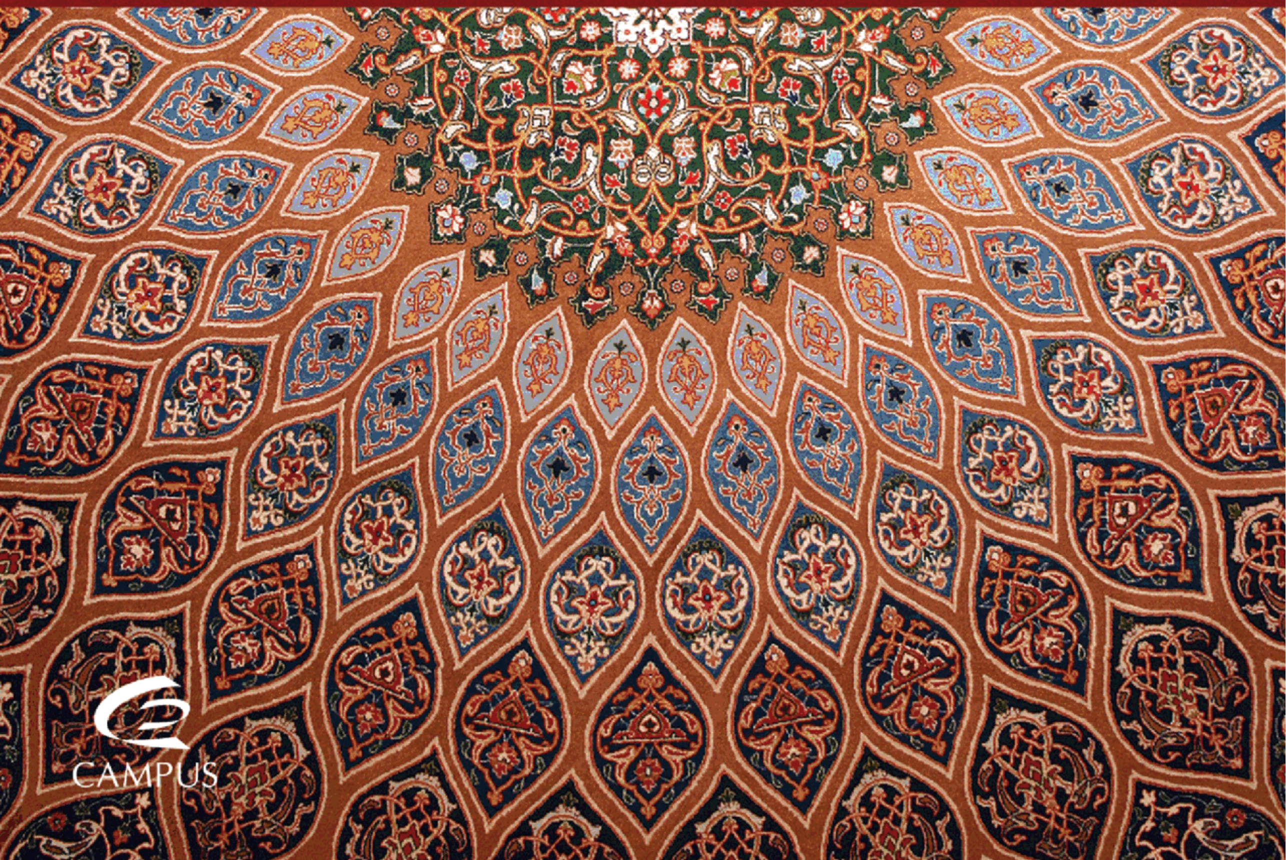
ELSEVIER

IVAN RICARTE

INTRODUÇÃO À COMPILAÇÃO



CAMPUS

IVAN RICARTE

INTRODUÇÃO À COMPILAÇÃO

ELSEVIER

CAMPUS

Copidesque
Ivone Teixeira

Editoração Eletrônica
Ivan Ricarte

Revisão Gráfica
Gypsi Canetti

Projeto Gráfico
Editora Campus/Elsevier
A Qualidade da Informação
Rua Sete de Setembro, 111 – 16º andar
20050-006 – Rio de Janeiro – RJ – Brasil
Telefone: (21) 3970-9300 Fax (21) 2507-1991
E-mail: info@elsevier.com.br
Escritório São Paulo
Rua Quintana, 753 – 8º andar
04569-011 – Brooklin – São Paulo – SP
Telefone: (11) 5105-8555

ISBN 978-85-352-3067-3

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.
Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

Central de atendimento
Tel.: 0800-265340
Rua Sete de Setembro, 111, 16º andar – Centro – Rio de Janeiro
E-mail: info@elsevier.com.br
Site: www.campus.com.br

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

R376I

Ricarte, Ivan
Introdução à compilação / Ivan Ricarte. - Rio de Janeiro : Elsevier, 2008.
il. ;

Contém exercícios
Inclui bibliografia
ISBN 978-85-352-3067-3

1. Compiladores (Computadores). I. Título.

08-1336. CDD: 005.453
CDU: 004.4'422

07.08.04 07.08.04 006084

Sumário

Prefácio

Há muitas décadas os computadores eram máquinas gigantescas e misteriosas, com quais o contato só era permitido a alguns poucos iniciados. Toda essa aura de mistério era amplificada pelas obras de ficção, e muitas atribuíam a esses "cérebros eletrônicos" características que faltam a vários seres humanos, como iniciativa própria e bom senso.

Com o avanço da tecnologia de integração eletrônica e o conseqüente barateamento dos componentes, o computador deixou de ser uma ferramenta cara e exclusiva dos grandes centros de pesquisas e passou a ser presença constante no nosso cotidiano. Ainda existem as grandes máquinas, mas a mesma tecnologia está presente nos computadores de mesa (*desktops*) e nos computadores portáteis (*laptops*, *notebooks*). Mas ela também marca presença em outros dispositivos, como em telefones celulares, PDAs (assistentes digitais pessoais) e em painéis de aparelhos eletrodomésticos.

Apesar dessa "quase onipresença" da tecnologia computacional, muitos ainda vêem na operação dos computadores algo de mágico. Um dos objetivos deste livro é desvendar parte desse mistério, no que diz respeito ao software que faz essa máquina funcionar. Particularmente, o foco deste texto está nas atividades relacionadas à compilação, ou seja, à tradução de especificações próximas ao nível de compreensão do ser humano para o nível de instruções que podem ser executadas pelos processadores.

Um dos primeiros pontos que precisam ser entendidos sobre o computador é o porquê de ele ser uma máquina tão especial, que a faz diferente de tantos outros engenhos inventados pela criatividade humana. Essencialmente, ela é especial porque é programável, ou seja, ela pode ser configurada para desempenhar diferentes tarefas sem ter de alterar substancialmente a sua configuração de circuitos. É essa flexibilidade que permite que basicamente a mesma tecnologia

seja usada para controlar operações simples em um painel eletrônico e também para controlar processos extremamente complexos.

A configuração básica dos circuitos eletrônicos de um computador recebe o nome genérico de hardware. É nessa configuração que se define a capacidade de operação de um computador — com qual velocidade ele irá operar e quão grandes são os problemas que ele poderá resolver. Eventualmente, parte dessa configuração pode ser alterada, por meio de uma troca de processador ou de uma expansão de memória, por exemplo, mas esses são eventos que não ocorrem freqüentemente.

A flexibilidade de um computador não vem da modificação de seus circuitos, mas sim das ordens ou instruções que são passadas para que esses circuitos executem suas tarefas. A partir de um conjunto básico de instruções, os computadores podem ser programados para realizar tarefas bem diversas. Essa componente mais flexível do computador recebe o nome de software, em contraposição à estrutura rígida do hardware do computador.

É por meio da programação habilitada pelo software que é possível, sobre um mesmo conjunto de circuitos, realizar operações tão distintas como escrever um texto, realizar cálculos em uma planilha ou navegar pela Internet. Esses diferentes tipos de programas, ou aplicativos, constituem a face mais conhecida dos computadores hoje em dia.

Os diferentes programas aplicativos são, evidentemente, os principais responsáveis pela popularização do uso de computadores, algo inimaginável na época em que eles ocupavam grandes salas nos centros de pesquisa e em grandes empresas. Mas não é apenas aí que o software está presente. O computador é uma máquina tão flexível que até o que faz com que esse software possa ser criado e possa funcionar também é programado — ou seja, também é software.

Para distinguir entre essas duas facetas do software, quando necessário, utilizamos alguns termos qualificativos. Assim, o software que é composto pelo conjunto de programas utilizados pelo usuário final, muitas vezes leigo em computação, é o chamado software de aplicação. Já o conjunto de programas que permite a criação e a operação do software de aplicação é denominado software de sistema.

O compilador é um dos programas do software de sistema, assim como o são o sistema operacional, os montadores, os carregadores e os ligadores — todos eles envolvidos na construção de aplicativos, conforme será visto neste texto. Mas as atividades associadas a um compilador aparecem também em outras aplicações que não apenas na geração de programas executáveis.

Onde há exemplos com programas existentes, a preferência foi dada sempre

à utilização de software disponível gratuitamente. Assim, os exemplos foram criados em sistema operacional Linux e com o compilador Gnu C++ (`g++`). No entanto, afora os detalhes das linhas de comando, os conceitos demonstrados independem dessa opção, e os mesmos exemplos podem ser reproduzidos em outros ambientes operacionais.

Uma boa parte deste livro foi desenvolvida no contexto das disciplinas Introdução a Software de Sistema e Mini e Microcomputadores: Software, oferecidas para os cursos de Engenharia de Computação e Engenharia Elétrica da Unicamp. Não seria justo deixar de expressar aqui a nossa gratidão aos alunos e docentes que ofereceram valiosos retornos ao adotar o texto preliminar nas suas disciplinas. Correndo o risco de deixar de citar outros colaboradores, quero expressar minha gratidão aos colegas Maurício Magalhães, Marco Aurélio Henriques e Eleri Cardozo, que sempre tiveram a paciência de sugerir melhorias no texto original e apontar as falhas presentes.

Não poderia também deixar de agradecer à equipe da Editora Campus-Elsevier, que muito colaborou para que este projeto chegasse à sua conclusão. Também com o risco de deixar de citar alguns nomes, quero agradecer a Ricardo Redisch, Silvia Lima, Marcia Henriques e André Gerhard Wolff.

Por fim, também quero agradecer à minha família, Núria, Laura, Sofia e Jordi, que compreendeu a minha distância nesse período em que o livro estava sendo preparado. A todos eles dedico este trabalho.

Capítulo 1

O compilador na visão do usuário

As aplicações computacionais estão cada vez mais presentes no cotidiano moderno, não apenas nos já tradicionais computadores de mesa mas também na forma de software embarcado nos mais diversos equipamentos, de aviões a eletrodomésticos. Afora as diferenças de capacidade e velocidade de processamento, os processadores presentes em todas essas aplicações têm essencialmente a mesma estrutura. O que traz essa possibilidade de diversificar o universo de aplicações é o desenvolvimento de software que é executado no processador.

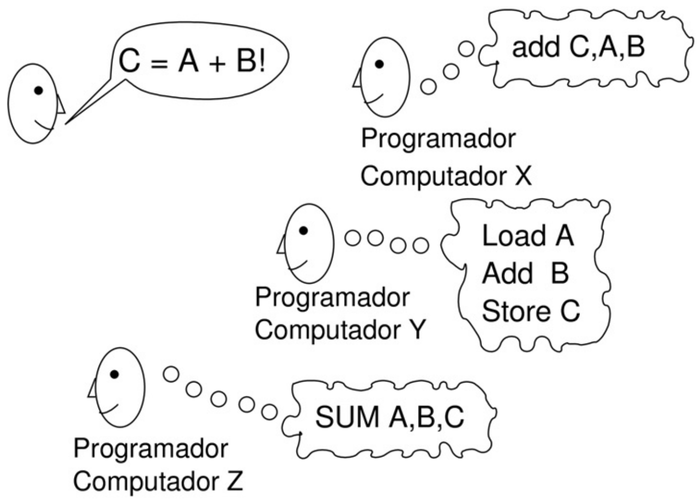
Este capítulo apresenta a visão que um usuário tem do programa que é essencial para obter esse grau de flexibilidade no desenvolvimento de software — o compilador. O restante do livro apresenta a estrutura e a forma de operação interna desse programa em detalhes.

1.1 Compiladores na programação

A grande motivação que catalisou o desenvolvimento de compiladores foi a demanda por uma maior eficiência de programação. O surgimento de computadores com diferentes arquiteturas e instruções distintas logo mostrou como era ineficiente o esforço de desenvolver aplicações para cada plataforma. Cada máquina era única, não apenas no seu conjunto ou jogo de instruções mas também na forma como os dados eram representados. Cada comando que fosse desejado na especificação do problema demandava que a implementação fosse desenvolvida de modo particular para cada máquina.

A situação de então é ilustrada na Figura 1.1, para o caso de uma única instrução de soma. De maneira muito simplificada, essa figura mostra bem qual era a situação real em meados da década de 1950, uma época em que ainda havia poucos computadores e nada que se aproximasse do conceito de uma família de processadores. Passar uma aplicação de uma máquina para outra demandava que uma nova implementação fosse realizada, em geral por um outro programador.

Figura 1.1 A programação antes dos compiladores

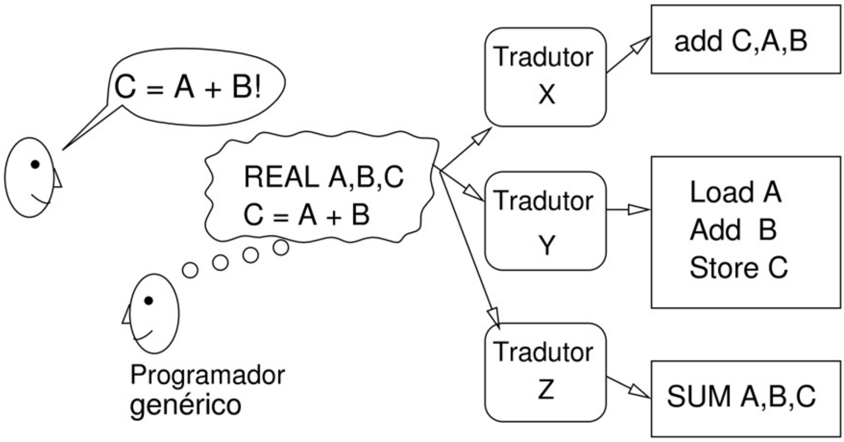


A situação ilustrada para uma única instrução nessa figura tinha sua complexidade muito aumentada quando se consideram que aplicações inteiras passavam por esse processo. Na ausência de linguagens comuns que permitissem expressar a especificação de forma independente da implementação na máquina, cada realização do código dependia da intervenção do humano que fosse especialista naquela plataforma. Claramente, uma situação pouco eficiente e de alto custo para as empresas.

A solução passava por buscar mecanismos que permitissem a "programação automática", ou seja, que traduzissem especificações genéricas, independentes de máquina, para código que pudesse ser efetivamente executado nos diferentes processadores. Essa proposta é ilustrada na Figura 1.2.

Pelo que se vê na figura, a proposta era ter uma única descrição de programa que estivesse suficientemente próxima da especificação original e que pudesse ser automaticamente traduzida para as diferentes máquinas, sem necessidade de intervenção humana. Desse modo, poderia haver um aumento de produtividade

Figura 1.2 A programação com os compiladores



e economia na produção. A maior produtividade seria decorrente do fato de o programador trabalhar com comandos de nível mais alto que a linguagem de máquina, sem ter de se preocupar com detalhes de instruções e operações intermediárias. A economia na produção decorreria do fato de que não seria mais necessário manter programadores para cada tipo de máquina.

Hoje, mais de cinqüenta anos depois dessa concepção que originou o desenvolvimento de compiladores, a situação ainda está longe de ser ideal. Há uma grande diversidade de linguagens de programação e ainda há especialistas numa ou noutra linguagem. Mas, certamente, sem compiladores o desenvolvimento da programação estaria num estágio muito mais primitivo do que o atual.

1.1.1 A linguagem dos processadores

Os processadores executam seqüências de comandos que fazem parte de um jogo de instruções definido pelos seus projetistas. Um programa de computador pode ser comparado a uma receita culinária, que indica os ingredientes e os passos elementares que devem ser seguidos para desempenhar uma tarefa. O processador, componente principal de um computador, é o responsável pela execução dessas instruções.

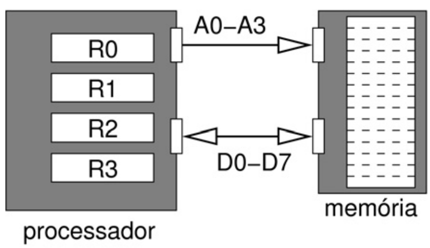
Cada processador tem seu jogo de instruções específico. A essas instruções estão associadas duas representações: a linguagem de máquina e a linguagem simbólica. A linguagem de máquina de um processador é representada pela

seqüência de bits (zeros e uns) que representa cada instrução, com sua operação e seus dados. O bit, um neologismo criado na língua inglesa para um dígito binário, ou seja, de apenas dois valores, é a unidade de informação utilizada pelos computadores digitais. Efetivamente, é pela manipulação de bits que o computador processa toda informação, por meio de sinais elétricos que assumem apenas dois níveis distintos. Em última análise, todo e qualquer programa que executa em um computador é representado por uma seqüência de instruções em linguagem de máquina, ou seja, uma seqüência de bits. É essa seqüência de bits que, passada ao processador, determina a ativação dos sinais que comandarão a execução da operação. No nível dos circuitos eletrônicos, a operação de um computador é a tradução (ou decodificação) dessas seqüências de bits em ações que devem ser desempenhadas pela CPU (a unidade central de processamento dos computadores) e circuitos adjacentes para a execução de cada instrução.

A linguagem simbólica, ou *assembly*, representa as mesmas instruções em formato textual, mais apropriado para a compreensão por seres humanos.

A título de ilustração, considere um processador hipotético representado de forma muito simplificada na Figura 1.3. Esse processador tem quatro registradores de dados, usados como área de armazenamento temporário para os dados envolvidos na execução das operações. Outros registradores que são essenciais para a operação do processador, como o registrador de instruções (que armazena a instrução que é executada), o contador de programas (que armazena o endereço da próxima instrução que deve ser executada) e o registrador de controle (que armazena o estado associado ao resultado da última execução), não são apresentados nessa figura. Esse processador é provavelmente um processador de 8 bits, pois essa é a dimensão do seu barramento de dados (bits D0 a D7). Sua capacidade de endereçamento de memória é de 16 posições, pois tem quatro linhas de endereço (bits A0 a A3); com quatro bits é possível representar $2^4 = 16$ valores diferentes.

Figura 1.3 Arquitetura de um processador hipotético



Considere que os projetistas desse processador tenham incorporado os circuitos para realizar quatro operações:

LOAD para transferir o conteúdo de uma posição de memória para um registrador. Por exemplo, a instrução `LOAD 10,R1` é usada para carregar o conteúdo da posição 10 de memória para o registrador R1;

STORE para transferir o conteúdo de um registrador para uma posição de memória. Por exemplo, a instrução `STORE R2,5` é usada para transferir o conteúdo do registrador R2 para a posição 5 de memória;

ADD para adicionar o conteúdo de dois registradores e armazenar o resultado em outro registrador. Por exemplo, a instrução `ADD R1,R2,R3` soma o conteúdo de R1 a R2 e coloca o resultado em R3;

BZERO para mudar o conteúdo do contador de programas para a posição de memória especificada se o conteúdo do registrador indicado for igual a zero. Por exemplo, a instrução `BZERO R3,7` define que o contador de programas será alterado para o valor 7 somente se o conteúdo de R3 for igual a zero.

Cada instrução desse processador precisa registrar qual é a operação que deve ser realizada. Como são quatro operações e cada bit pode assumir dois valores, são necessários dois bits (pois $2^2 = 4$) para representar cada código de operação. Neste exemplo, imagine que os projetistas designaram os códigos 00 para LOAD, 01 para STORE, 10 para ADD e 11 para BZERO. Além do código de operação, é preciso registrar na instrução quais são os operandos envolvidos. Para tal fim, são usados quatro bits para representar uma posição de memória e dois bits para identificar o registrador (por exemplo, 00 para R0, 01 para R1, 10 para R2 e 11 para R3). Como os operandos são três registradores para a operação ADD ou um registrador e uma posição de memória para as demais operações, seis bits são necessários para representar os operandos. Assim, uma instrução pode ser completamente representada por uma seqüência de oito bits.

Considere o seguinte programa, representado inicialmente em linguagem simbólica, que soma o conteúdo da posição 10 com o conteúdo da posição 11 e armazena o resultado na posição 12:

```
LOAD  10, R1
LOAD  11, R2
ADD   R1, R2, R0
STORE R0, 12
```

O código que deve estar armazenado na memória para que o processador possa executar o programa é o código em linguagem de máquina equivalente. Por exemplo, para a instrução `LOAD 10,R1`, o código em linguagem de máquina começa com a seqüência de bits que identifica a operação, 00; é seguido pelo endereço de memória, valor 10 em decimal, cuja representação binária é 1010; e termina com a identificação do registrador, R1, identificado pelo valor binário 01. Assim, o código completo da instrução é 00101001.

Da mesma forma, pode-se obter o código para as demais instruções. Desse modo, o programa tem a seguinte representação equivalente em código de máquina:

```
00101001
00101110
10011000
01001100
```

Claramente, seria muito desconfortável e pouco produtivo se os programadores tivessem de desenvolver cada uma de suas aplicações escrevendo o código em linguagem de máquina. Os programadores não trabalham diretamente com a representação binária das instruções de um processador, mas sim com representações simbólicas mais abstratas. A linguagem simbólica oferece um primeiro nível de abstração ao representar as mesmas instruções por meio de textos. É muito mais fácil para um ser humano escrever e entender um programa com instruções na forma textual, como `MOVE 11,R2`, do que com o código binário equivalente, como `01101110`. Já para o processador, o formato binário é o formato natural.

Embora seja possível que programadores desenvolvam software utilizando apenas a linguagem simbólica de um processador, esse não é o procedimento usual. Dois são os principais motivos para que isso não ocorra. O primeiro é o baixo nível de abstração dessa linguagem. Apesar de ter um nível de abstração maior que a linguagem de máquina, a linguagem simbólica ainda representa as operações e características de cada processador de forma diferente. Por ainda estar tão próxima do nível dos circuitos, qualquer operação um pouco mais complexa demandaria um grande esforço de programação, com grande possibilidade de introdução de defeitos no código. O segundo motivo é a falta de portabilidade — passar uma aplicação já desenvolvida para um processador para um outro processador exige que todo o programa seja codificado novamente, na nova linguagem. Por outro lado, por permitir que o programador trabalhe diretamente com as instruções do processador, a linguagem simbólica é usualmente utili-

zada para a programação de fragmentos de código nos quais é essencial obter uma alta velocidade de execução.

1.1.2 Linguagens de alto nível

A alternativa para solucionar os problemas associados à programação em linguagem simbólica é criar as descrições dos programas com uma linguagem de alto nível. Essas linguagens têm nível de abstração maior que as linguagens simbólicas — maior nível de abstração significa menos detalhes e, nesse caso, isso traduz-se em independência em relação ao processador. Quando o programador desenvolve uma aplicação com uma linguagem como Pascal, C ou Java, ele não se preocupa de antemão em saber qual é o processador no qual o programa será executado.

Em linguagens de alto nível, as instruções são expressas na forma de um texto que independe do processador. Esse texto, o código-fonte, é composto por uma combinação de palavras e expressões que se aproximam mais daquelas usadas na linguagem humana do que daquelas usadas para comandar o processador. O objetivo é facilitar, para os programadores, as tarefas de expressar os comandos de um novo programa e de compreender os programas existentes.

Por razões históricas, a língua inglesa foi adotada como a linguagem humana da qual os comandos das linguagens de computador deveriam se aproximar. Por esse motivo, usualmente os comandos de uma linguagem de alto nível têm nomes como `if` ou `while`, termos em inglês para denotar, respectivamente, alternativa (se) e repetição (enquanto).

Várias linguagens de alto nível estão disponíveis em diversas máquinas distintas. Por exemplo, a linguagem de programação C foi criada para execução em um minicomputador da Digital Corporation, modelo PDP-8, com o sistema operacional Unix. Essa mesma linguagem é utilizada atualmente no desenvolvimento de aplicações para computadores pessoais, para dispositivos portáteis, para computadores de grande porte e para sistemas de processamento paralelo, com uma grande diversidade de sistemas operacionais. Afora os recursos ou limitações inerentes a cada plataforma, a mesma linguagem C é utilizada para o desenvolvimento em todas essas plataformas.

Em geral, cada linguagem de alto nível teve uma motivação diferente para ser desenvolvida. Basic foi desenvolvida para ensinar princípios de programação; Pascal, para o ensino de programação estruturada; Fortran, para aplicações em computação numérica e científica; lisp e Prolog, para aplicações em inteligência artificial; Java, para o desenvolvimento de software embarcado e distribuído; C e C++, para a programação de sistemas.

No entanto, os processadores não entendem essas linguagens de alto nível. Vários esforços foram feitos para criar processadores que pudessem interpretar diretamente instruções de uma linguagem de alto nível, mas a conclusão sempre foi que os processadores são muito mais eficientes na execução de código em linguagem de máquina. Assim, é preciso que a descrição de um programa numa linguagem de alto nível seja traduzida para o código em linguagem de máquina do processador que irá executar o programa. O importante é que essa tradução seja bem feita, para que o código gerado seja eficiente. Esse processo de tradução de um código de computador de um formato para outro ocorre muitas e muitas vezes, em alguns casos até mesmo de forma transparente para o usuário. Assim acontece quando um programa em C, por exemplo, é transformado em um código executável expresso em linguagem de máquina.

O compilador é o programa de sistema que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em outra linguagem, como o código de máquina para um processador. Programas de sistema são aqueles que fazem com que os outros programas funcionem. Esses outros programas podem ser outros programas de sistema, como sistemas operacionais, montadores, carregadores, ligadores, ou podem ser programas de aplicação, como editores, planilhas e bancos de dados. Enquanto programas de aplicação têm como dados textos ou números, programas de sistema manipulam outros programas como seus dados.

O primeiro compilador completo para uma linguagem de programação foi criado em meados da década de 1950 para a linguagem Fortran [Backus, 1998]. No entanto, muitas das ferramentas ainda hoje em uso para a construção de compiladores e de tradutores de linguagens de uma forma geral foram criadas no escopo do desenvolvimento do sistema operacional Unix e da linguagem de programação C, no início da década de 1970. Desde então, essas ferramentas ganharam implementações mais eficientes e foram disponibilizadas para outras linguagens de programação.

Para ilustrar como um programa de alto nível isola os detalhes do processamento envolvido na execução, as quatro instruções do programa em linguagem simbólica da seção anterior seriam provavelmente o resultado de uma linha de um programa em linguagem de alto nível — por exemplo, em C:

```
c = a + b;
```

Como em C todas as variáveis usadas devem ser declaradas, na verdade o programa deveria conter também uma linha com essas declarações antes da linha com a soma:

```
int a, b, c;
```

O que ocorre na tradução dessas linhas de código em linguagem C para o código em linguagem simbólica é um processamento que envolve a leitura do programa em linguagem de alto nível (o código-fonte) e a escrita de um programa equivalente em linguagem simbólica. Entre ler um programa e escrever o outro, o compilador terá de desempenhar muitas tarefas: criar uma representação interna a partir do programa de entrada, verificar a correção dos comandos, verificar a aderência às regras da linguagem e transformar a representação interna num formato adequado à saída desejada.

1.2 Compiladores no processamento da informação

O que está envolvido na atividade de um compilador é, de maneira ampla, a tradução de uma especificação em outra. Apesar da origem no desenvolvimento de programas executáveis, as mesmas tarefas que um compilador realiza para traduzir um programa em linguagem de alto nível para um programa na linguagem de máquina são também realizadas para processar informação de outra natureza. Um exemplo cada vez mais presente é a transformação de representações de informação de um formato para outro. Mesmo que nem sempre essas transformações recebam o nome de "compilação", não é difícil perceber como há similaridades entre elas.

1.2.1 Processamento de arquivos XML

Uma aplicação cada vez mais presente no cotidiano de quem usa computadores, principalmente na navegação pela World Wide Web, é o processamento de informações em arquivos na linguagem XML (*Extensible Markup Language*). De forma transparente para o usuário final, arquivos XML são utilizados para troca de informações entre clientes e servidores e constituem a base tecnológica para a disponibilização de serviços por meio da Web. Em outros contextos são muito utilizados para armazenar informações de configuração de aplicações e também para representar textos estruturados.

A informação em um arquivo XML, embora possa ser lida por seres humanos, está lá para ser processada e traduzida para outro formato — para a execução de uma tarefa num servidor, para definir a interface de uma aplicação ou para produzir um texto que possa ser impresso, por exemplo. Em todas essas situações, a tradução de uma especificação genérica para um formato que possa ser utilizado pelo computador está presente e remete às mesmas atividades presentes na compilação de um programa.

Essencialmente, o conteúdo de um arquivo XML está organizado em uma hierarquia de elementos, com cada elemento delimitado por marcações. Marcações em XML são reconhecidas por estarem entre colchetes angulados, <>. Por exemplo, o seguinte fragmento de informação poderia ser encontrado num arquivo XML contendo dados sobre livros:

```
<livro>
  <titulo>Dom Casmurro</titulo>
  <autor>Machado de Assis</autor>
  <ano>1900</ano>
</livro>
```

Nesse fragmento há um elemento `livro`, que tem em seu conteúdo três outros elementos. Esses elementos, por sua vez, têm texto como conteúdo. Elementos podem conter outros elementos, texto, ambos ou ainda ser vazios.

Esse é um exemplo de informação estruturada. Apesar de ser evidente para um ser humano qual é a informação que está lá presente, a informação está lá para ser processada por uma máquina. Um dos atrativos de XML, que é responsável por boa parte da sua popularidade entre muitas aplicações, está exatamente no fato de que a estrutura é genérica o suficiente para servir para qualquer aplicação. Ela pode ser utilizada para alimentar a informação em um banco de dados, pode também ser utilizada para criar um índice impresso ou ainda para gerar uma página a ser exibida na Web.

Todo arquivo XML deve obedecer a algumas regras sintáticas fundamentais, independentemente de qual seja a aplicação ou a interpretação que será feita do arquivo. São as chamadas regras de boa formação de XML:

1. Todo documento XML deve ter um único elemento que sirva como raiz para todos os demais elementos do documento;

2. Todo elemento deve ter a marcação de início e a marcação de final; elementos vazios podem ser representados com uma única marcação na forma `<.../>`;

3. Dois elementos não podem estar entrelaçados no documento;

4. Elementos podem ter atributos, representados entre aspas na marcação de início do elemento;

5. Nomes de elementos devem começar com letras ou o caractere de sublinhado e podem conter, além deles, números, hifens e pontos.

Um analisador de conteúdos XML, o equivalente a um compilador para esse tipo de arquivo, analisa pelo menos se o conteúdo do arquivo obedece a essas regras.

Além dessas regras básicas e genéricas para qualquer arquivo em formato XML, é possível ter uma descrição de como os elementos de um arquivo específico para um tipo de aplicação devem estar organizados. Isso é necessário quando o computador precisar realizar alguma ação a partir do conteúdo do arquivo XML — é preciso garantir que toda a informação necessária para essa realização esteja corretamente especificada. Nesse caso, o analisador pode realizar, além da verificação sintática básica, uma verificação de que o documento está de acordo com esse modelo.

Nesse aspecto, o processamento de um arquivo XML aproxima-se muito da compilação. Há uma fonte única para os dados — naquele caso, um programa em linguagem de alto nível; aqui, informações no formato XML. O processamento que é realizado com esse arquivo envolve a análise do conteúdo, para verificar sua aderência às regras da linguagem, e pode gerar saídas diferenciadas — na compilação de programas, arquivos com códigos em linguagem de máquina que podem ser executados em diferentes processadores; no processamento XML, transformações ou ações de acordo com a aplicação. Para que isso seja possível, é preciso que a aplicação leia o arquivo XML, verifique que ele está correto, crie uma representação interna da informação que ele contém e possa gerar a saída no formato desejado. Não muito diferente do que faz um compilador para uma linguagem de programação.

1.2.2 Páginas dinâmicas na World Wide Web

Páginas na World Wide Web são disponibilizadas a partir de um servidor, que atende a solicitações de clientes — tipicamente, um programa navegador Web no qual o usuário selecionou uma página para exibição, seja pela especificação de um endereço, seja pela seleção de uma ligação (um *link*) numa página já exibida. Quando o servidor tem a página com seu conteúdo já definido, seu

único trabalho é ler o arquivo correspondente do seu disco e transferi-lo para o cliente que fez a solicitação.

Páginas dinâmicas, por outro lado, não têm esse conteúdo pronto. O servidor deve criar esse conteúdo a partir de um gabarito, ou seja, de uma página com lacunas que devem ser preenchidas no momento em que o usuário solicita a página. Embora inicialmente não fosse assim, atualmente o processamento de páginas dinâmicas na Web pode ser visto como um caso particular do processamento XML. Em vez de comandos genéricos definidos para cada implementação de servidor, o usual atualmente é utilizar uma linguagem de marcação baseada em XML para definir ações que devem ser executadas para preencher as lacunas.

Um exemplo de uma linguagem desse tipo é JSP (*Java Server Pages*), que trabalha em um servidor que permite a execução de comandos na linguagem Java. No momento da execução, no atendimento a uma solicitação do cliente, o processamento do elemento XML que corresponde a um elemento JSP dá início a um processamento, cujo resultado é agregado à página que está sendo gerada. O resultado final é uma página em HTML (*Hypertext Markup Language*), a linguagem apropriada para exibição de conteúdos em um navegador Web.

Considere, por exemplo, como seria a criação de uma página dinâmica especificada em uma linguagem hipotética, a CSP (de *C Server Pages*). A estrutura de um arquivo desse tipo é tipicamente:

```
<csp>
<@ #include ... %>
<% // código C ... %>
<html>
      Conteúdo em HTML
      com <%= código C %>
</html>
</csp>
```

Como um documento XML, essa especificação obedece a todas as regras de boa formação. A página CSP contém diretivas que são usadas para criar um programa em C++ que gera a resposta, em HTML, para a solicitação a ela encaminhada. Essas diretivas podem ser instruções para o pré-processador C, entre `<@` e `%>`, que são incluídas no início do arquivo-fonte gerado; código-fonte em C a ser inserido no arquivo-fonte, entre `<%` e `%>`; texto a ser repassado literalmente para a página HTML a ser gerada, exceto por fragmentos de código que podem ser utilizados para obter valores de variáveis, entre `<%=` e `%>`.

Como se pode observar, também nesse caso há uma tradução de uma especificação com um alto nível de abstração (as diretivas da linguagem) para uma representação equivalente, de nível de abstração mais baixo (o código em C++) e que pode ser interpretada pela máquina — nesse caso, compilada e depois executada para gerar a resposta em HTML.

1.3 Atividades de um compilador

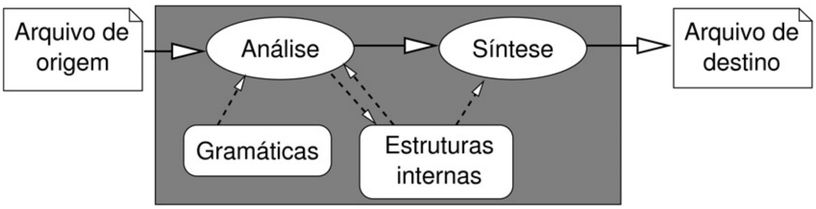
Independentemente de qual seja a linguagem de alto nível que o compilador irá traduzir e de qual seja o processador que é o alvo dessa tradução, todos os compiladores executam algumas tarefas comuns.

Nas aplicações de programação, o compilador recebe como entrada um arquivo com uma especificação em linguagem de alto nível, o código-fonte, e gera como saída um outro arquivo com o programa equivalente em linguagem simbólica do processador da plataforma de destino. Nesse caso, um outro programa, o montador (ou *assembler*), é utilizado para traduzir o código em linguagem simbólica para o código de máquina.

Nos outros tipos de aplicações apresentados, também há a necessidade de tarefas similares — a leitura e o reconhecimento de formato de um arquivo de entrada e a geração de uma informação equivalente num outro formato.

A Figura 1.4 expande essa visão um pouco mais, ao apresentar algumas características internas da operação de um compilador.

Figura 1.4 Estrutura interna de um compilador



A entrada para o compilador é um arquivo de texto, que será lido caractere a caractere para compor a estrutura do programa, uma descrição interna do compilador que permitirá a geração de um código equivalente. A criação dessa estrutura é resultado da etapa da análise do código de origem. Durante a análise, pelo menos duas tarefas básicas são realizadas e, em geral, tratadas de forma separada.

A primeira tarefa na análise é a identificação, a partir dos caracteres individuais, dos símbolos básicos válidos para a linguagem. Em uma linguagem de programação, são os nomes de comandos, nomes de variáveis, nomes de funções, constantes, delimitadores. Em um arquivo XML, são os elementos e conteúdos textuais. Essa etapa é denominada análise léxica, e é apresentada em detalhes no Capítulo 3.

Com o resultado da análise léxica, o compilador tenta compor, a partir da estrutura de cada um dos blocos básicos, a estrutura geral do arquivo de entrada. Em uma linguagem de programação, o objetivo é reconhecer a estrutura do programa a partir de declarações, definições de funções e suas seqüências de comandos simples ou compostos. Em um arquivo XML, o objetivo é reconhecer a hierarquia de elementos que compõem a informação completa. Analogamente ao que se faz na interpretação de um texto em linguagem natural, nessa etapa o objetivo é entender a estrutura das frases a partir da combinação das palavras individuais. Por tal analogia, essa etapa, apresentada no Capítulo 4, é conhecida como análise sintática.

A etapa de análise é concluída com a realização da análise semântica, assunto do Capítulo 5. Como será visto, essa atividade de um compilador tem objetivos bem mais modestos que a correspondente atividade em linguagens naturais e busca verificar a coerência entre os fragmentos de códigos sintaticamente corretos.

A analogia de um programa com um texto em linguagem natural não se encerra nos nomes dessas etapas da análise. Para que essas duas etapas possam ser realizadas, é preciso que o compilador conheça as regras da linguagem. Como saber se um símbolo ou nome de elemento é válido ou não? Como saber se a ordem dos elementos em um comando ou elemento está correta? Para tal fim, o compilador utiliza descrições da linguagem organizadas de acordo com uma gramática, com as regras que descrevem o que é válido. Há gramáticas apropriadas para a análise léxica e gramáticas apropriadas para a análise sintática. Gramáticas são descritas em detalhes no Capítulo 2.

A partir do momento em que uma estrutura correta é reconhecida, é possível gerar a tradução dessa estrutura na linguagem de destino. Essa tradução não é feita diretamente a partir da expressão original, mas sim a partir da representação da estrutura da expressão que foi construída na análise. O que será feito a partir do reconhecimento dessa estrutura claramente depende da aplicação.

Em uma aplicação de programação, essa atividade de síntese do arquivo de saída, que pode ser tanto na forma de um outro arquivo texto ou um arquivo binário, preparado para ser combinado com outros, engloba normalmente duas

etapas. A primeira é a geração de código a partir da estrutura que foi criada na análise. Como esse código é criado a partir das estruturas elementares, é possível que haja no código gerado, na combinação dos segmentos de código básico, muitas possibilidades de melhoria. Tais melhorias são buscadas na segunda etapa da síntese, que é a otimização de código. Essas etapas da síntese são apresentadas no Capítulo 6.

1.3.1 Leitura de arquivo de origem

Antes de realizar qualquer tarefa de interpretação dos elementos do arquivo de entrada, o compilador precisa ler os caracteres de entrada que estão num arquivo de texto, armazenado no disco. Esta seção apresenta os mecanismos oferecidos pela linguagem C++ para realizar essa tarefa básica.

A manipulação de arquivos de texto em C++ é suportada pelo conceito de *streams* de dados. O termo stream denota um fluxo seqüencial, como aquele seguido pelas águas de um riacho. Na computação, um stream pode ser uma fonte de dados, a partir da qual o programa recebe um elemento de cada vez, ou pode ser um receptor de dados, para o qual o programa pode enviar um elemento de cada vez. No processamento de dados binários, o elemento de transferência entre programa e streams é um byte; para arquivos de texto, é um caractere.

Para permitir a entrada ou saída de dados a partir de um arquivo no disco, a linguagem C++ especifica um conjunto de classes no arquivo de cabeçalho `fstream`. Os recursos para a leitura de dados a partir de um arquivo são providos pela classe `ifstream`. Assim, o código de um programa C++ que manipula dados em arquivos deve conter a diretiva:

```
#include <fstream>
```

O arquivo a partir do qual os dados serão obtidos estará associado no código a um objeto da classe `ifstream` — um objeto pode ser visto, *grosso modo*, como uma variável cujo tipo é uma classe. Há duas formas de especificar o nome desse arquivo de forma a associá-lo ao objeto `ifstream` correspondente. A primeira é por meio de um argumento na própria declaração do objeto; no jargão de C++, pela utilização do construtor com argumentos para objetos dessa classe. Nesse caso, a linha de código que define o objeto (`arq`, neste exemplo) tem também o nome do arquivo (por exemplo, `learq.cpp`):

```
ifstream arq("learq.cpp");
```

A outra maneira é pela aplicação do método `open` ao objeto `ifstream` previamente declarado. Nesse caso, declaração e indicação do nome do arquivo a ser aberto estão em comandos executados em momentos diferentes, como em

```
ifstream arq;
...
arq.open("learq.cpp");
```

Em qualquer das duas situações, a string constante (o texto entre aspas no código) poderia ter sido substituída por uma variável com o nome do arquivo.

Uma vez identificado e aberto o arquivo sobre o qual as operações serão realizadas, o próximo passo é realizar as operações de leitura. Para arquivos de texto, podem ser utilizados o operador >>, o método `get` ou o método `getline`.

O operador >> faz a leitura a partir de um stream de entrada para uma variável. Por exemplo, a linha

```
char ch;
...
arq >> ch;
```

lê um caractere do arquivo associado ao objeto à esquerda do operador, `arq`, e armazena-o na variável indicada à sua direita, `ch`.

Esse operador realiza a interpretação de formato, ou seja, realiza a conversão do formato texto para a representação interna de variáveis da linguagem. A representação textual (formatada) de um valor inteiro é uma seqüência de caracteres, que é diferente da representação interna de valores inteiros no computador, tipicamente de 32 bits em complemento de dois. O operador >> realiza automaticamente essa conversão, se necessário, de acordo com o tipo de variável que está à sua direita. Para o uso em compiladores, tipicamente serão lidos caracteres do arquivo.

Outro método importante na leitura de arquivos é `eof`, que retorna um valor lógico verdadeiro quando o final do arquivo for alcançado. Por exemplo, a estrutura básica de um programa para ler o conteúdo de um arquivo do início ao fim, caractere a caractere, é tipicamente:

```
#include <fstream>
//...
  ifstream arq;
  char ch;
```

```
//...
arq.open(...);
while (! arq.eof()) {
   arq >> ch;
   // processa ch
}
```

No entanto, é preciso ressaltar que o operador de leitura >> tem como comportamento padrão ignorar os chamados "espaços em branco": o espaço, o caractere de tabulação (`'\t'`, na representação de constantes do tipo caractere em C) e o caractere de mudança de linha (`'\n'`). Se for importante para o processamento reconhecer esses caracteres, esse operador não pode ser utilizado para a leitura a menos que esse comportamento padrão seja alterado.

Isso pode ser feito por meio dos métodos de alteração da especificação de formato de entrada e saída, `setf` e `unsetf`. Nesse exemplo, o que se deseja é retirar (*unset*) a característica de ignorar espaços em branco (*white spaces*). Em C++, essa característica está associada ao especificador `ios::skipws`. Assim, a inclusão de uma linha com o comando

```
arq.unsetf(ios::skipws);
```

antes do laço de leitura dos caracteres fará com que todos os espaços, tabulações e quebras de linhas sejam preservados mesmo que o operador >> seja utilizado. Assim, esses caracteres (`' '`, `'\t'` e `'\n'`, respectivamente) serão lidos para a variável `ch` quando encontrados, e a aplicação pode processá-los quando necessário.

Essa é uma característica importante no processamento de arquivos de origem, pois esses caracteres tipicamente são utilizados para separar os elementos das sentenças de entrada. Cada uma dessas palavras, ou *tokens*, precisa ser reconhecida e tratada isoladamente antes da construção da estrutura interna que representa toda a entrada.

Outra forma básica de realizar a leitura do arquivo de origem utiliza o método `get`, que não faz interpretação de formato dos caracteres de entrada. Nesse caso, uma instrução para leitura de um caractere do arquivo é tipicamente:

```
arq.get(ch);
```

Nesse caso, os separadores (espaços em branco, tabulações e mudanças de linha) não são interpretados de forma especial e o programa tem como reconhecer a sua ocorrência, independentemente da alteração do especificador `skipws`.

A terceira alternativa de leitura, menos utilizada no processamento de arquivos para compiladores, é a de obter vários caracteres em um bloco — tipicamente uma linha completa. Para tanto, o programador pode utilizar o método `getline`, com a indicação de uma área para o armazenamento do bloco lido (um arranjo de caracteres) e de quantos caracteres podem ser lidos de uma vez (limitado pela dimensão do arranjo). Um terceiro argumento opcional indica um terminador para a leitura, ou seja, um caractere que, se encontrado, termina o bloco mesmo que a quantidade de caracteres especificada ainda não tenha sido lida. Tipicamente, para a leitura de linhas, esse terminador é o caractere `'\n'`. Por esse motivo, esse é o valor implicitamente assumido pelo método se esse terceiro argumento for omitido.

Para os compiladores, o arquivo de entrada é normalmente um arquivo em formato texto. Outros aplicativos do sistema que manipulam arquivos com código de máquina, como o programa ligador e o carregador, por exemplo, precisam ler arquivos em formato binário. Em C++, essa especificação pode ser feita no momento da abertura do arquivo com a indicação de um segundo argumento, que corresponde ao modo de operação. Para a leitura de arquivos com conteúdo binário, cujos conteúdos serão bytes não necessariamente associados a caracteres, o especificador `ios::binary` deve ser utilizado. Para leitura de arquivos no formato texto, considerado como o padrão, não há nenhuma especificação de modo de operação a associar.

1.3.2 Escrita de arquivos de destino

Outra tarefa que o compilador deverá realizar é a criação de um arquivo de destino, que será utilizado por outros aplicativos do sistema para concluir o processo de transformação do código de entrada. Para um compilador de uma linguagem de programação, por exemplo, esse seria o arquivo com o programa equivalente em linguagem simbólica. Tipicamente, esse programa estará num arquivo temporário armazenado em disco e que será eliminado quando o montador escrever o arquivo com o código binário equivalente.

Para escrever dados da aplicação para um arquivo (stream) em disco, o programador deve utilizar um objeto da classe `ofstream`. Essa classe, assim como a classe `ifstream`, é especificada no arquivo de cabeçalho `fstream`.

Também da mesma forma que para a leitura de arquivos, o nome do arquivo no qual ocorrerá a escrita pode ser especificado junto à declaração do objeto, como um argumento para o seu construtor, ou em um momento posterior, com o uso do método `open` à variável do tipo `ofstream` declarada previamente.

A saída de dados para o arquivo é realizada pelo operador <<. A implementação desse operador pelos projetistas da linguagem C++ permite que ele reconheça diversos tipos de argumento, como os vários tipos primitivos da linguagem. De forma análoga àquela do operador de leitura >>, esse operador realiza o processo de formatação dos dados, ou seja, de acordo com o tipo de dado ele gera uma representação em formato textual adequada para um arquivo em formato texto ou para exibição na tela.

Manipuladores predefinidos podem ser utilizados para alterar o padrão de formatação para a saída. O alinhamento dos caracteres formatados pode ser alterado com os manipuladores `right` ou `left`. Por exemplo, para alinhar o valor inteiro da variável `val` à direita numa linha de saída do arquivo `arq`, o seguinte fragmento de código poderia ser utilizado:

```
arq << right << val;
```

Para valores inteiros, o comportamento padrão é apresentar a seqüência de caracteres correspondente à representação decimal do valor. É possível alterar o padrão com os manipuladores `hex`, para obter a representação hexadecimal, ou `oct`, para a representação em octal. No mesmo exemplo, para apresentar o valor em formato hexadecimal, o manipulador `hex` é utilizado:

```
arq << right << hex << val;
```

O comportamento padrão pode ser restaurado, após qualquer modificação, com o manipulador `dec`.

As operações de escrita, por razões de eficiência, não são realizadas diretamente no arquivo em disco. Internamente, os dados são primeiro armazenados em uma área interna (um *buffer*) para depois serem transferidos em bloco para o arquivo em disco. Em geral, o momento dessa transferência é controlado pelo sistema. Quando o programador quiser explicitar o momento no qual o conteúdo do buffer deve ser transferido, o método `flush`, sem argumentos, deve ser utilizado. Quando o modificador `endl` é utilizado, além de incluir o caractere `\n` na saída, o método `flush` é invocado:

```
arq << right << hex << val << endl;
```

Nas aplicações de compilação, tipicamente o arquivo de destino está no formato texto, que é o padrão assumido em C++. Para outros aplicativos, como o montador ou o ligador, por exemplo, esses arquivos devem estar no formato

binário. Um segundo argumento no método de abertura do arquivo, além do nome do arquivo a ser aberto, indica o modo de operação. A especificação `ios::binary` é utilizada para escrever conteúdos binários em vez de caracteres em formato texto.

Saída sem interpretação de características de formatação pode ser realizada com o método `put`, que recebe um caractere como argumento. Também é possível escrever um bloco de dados em vez de caracteres individuais. Nesse caso, o método `write` é utilizado. Este recebe dois argumentos: o ponteiro para o início do bloco que contém os caracteres e a quantidade de caracteres que deve ser transferida para a saída. Como `put`, não há interpretação de formato.

Outros especificadores são utilizados para indicar como o conteúdo de um arquivo já existente deve ser tratado. Se o novo conteúdo deve ser agregado ao final do conteúdo existente, então o especificador `ios::app` é utilizado. Se o conteúdo existente tiver de ser substituído pelo novo conteúdo, o especificador `ios::trunc` é utilizado.

1.3.3 Interação com arquivos padrão

Todo programa, ao executar, está habilitado a interagir com o "mundo exterior" por meio de três canais, que são denominados arquivos padrão do sistema. Os dispositivos padrão de interação com o usuário são o teclado (dispositivo de entrada padrão) e a tela do monitor (dispositivo de saída padrão). O primeiro desses canais é o arquivo padrão de entrada, que está inicialmente associado ao teclado para permitir a recepção de caracteres de forma interativa durante a execução da aplicação. Os outros dois canais estão associados à apresentação de textos na tela. Um deles está associado ao envio de mensagens durante a execução (saída padrão) e o outro está associado ao envio de mensagens de erros (saída padrão de erros).

Em C++, esses arquivos padrão são representados por três streams. Um desses streams é `cout`, para o arquivo padrão de saída. Os outros streams padrão são `cerr`, para a saída de mensagens de erro, e `cin`, para a entrada de caracteres a partir do teclado. Inicialmente, `cout` e `cerr` apresentam as mensagens na mesma saída, mas podem eventualmente ser separados. Para utilizá-los, é preciso incluir no início do arquivo com o código-fonte as linhas

```
#include <iostream>
using namespace std;
```

A primeira linha é uma instrução para o pré-processador da linguagem C,

indicando que o arquivo de definições `iostream` deve ser incorporado ao código. É nesse arquivo que se encontram as definições para a utilização dos arquivos padrão de entrada e saída. A outra linha refere-se ao conceito de *namespace*; em C++, o programador pode associar um prefixo aos identificadores que ele define. A biblioteca padrão de C++ utiliza o prefixo `std` para suas definições. O comando `using` indica que esse prefixo será utilizado para alguns dos símbolos na seqüência. Se esse comando for omitido, cada referência a um desses símbolos deverá conter o prefixo de *namespace* `std::`.

A entrada padrão está associada ao objeto `cin`, que é da classe `istream`. Assim como para a leitura de dados de um arquivo em disco, é possível utilizar para a leitura o operador de entrada com formatação de dados, >>. Por exemplo, em

```
int a;
cin >> a;
```

espera-se que o usuário digite no teclado uma seqüência de caracteres numéricos que será traduzida para um valor inteiro, o qual é armazenado na variável indicada.

É possível também utilizar a entrada padrão para obter entradas não-formatadas, ou seja, obter as entradas na forma de um caractere ou uma seqüência de caracteres sem interpretação. Para tanto, os métodos `get` e `getline` são utilizados. Por exemplo, para ler do teclado uma linha de até 80 caracteres, uma seqüência de expressões como a seguinte estaria no programa:

```
const int tamanhoLinha = 80;
char area[tamanhoLinha];
cin.getline(area, tamanhoLinha);
```

As saídas padrão estão associadas aos objetos `cout` e `cerr`, ambos instâncias da classe `ostream`. O operador de saída formatada de dados para essa classe é <<, o qual traduz o conteúdo da variável indicada para o formato textual adequado à apresentação nessas saídas. Por exemplo,

```
int a;
...
cout << "Valor de a = " << a << endl;
```

apresenta na saída padrão a mensagem indicada, seguida pela representação textual do valor da variável `a`.

1.4 Exemplos de compiladores

Nesta seção serão apresentados dois exemplos de compiladores. O primeiro é um compilador tradicional, para uma linguagem de programação — o compilador `g++` para a linguagem C++. O outro exemplo ilustra uma aplicação com processamento de arquivo em XML.

1.4.1 Compilador C++

O compilador `g++` é parte da coleção de compiladores Gnu, da Free Software Foundation. Tipicamente, para aplicações simples, ele toma um ou mais arquivos de entrada como argumentos e produz um arquivo executável de saída, caso tudo esteja correto. Dessa forma, observa-se que além da compilação propriamente dita esse compilador executa, de forma transparente para o usuário, outros aplicativos necessários para a criação do programa executável, como o montador e o ligador.

Apenas para ilustrar o uso desse compilador, será utilizado o clássico programa *hello, world*. Esse programa foi publicado pela primeira vez no livro original da linguagem de programação C e desde então foi utilizado como exemplo elementar de programa em diversas linguagens de programação — nem sempre tão básico, de acordo com a linguagem de programação escolhida. Na versão em C++, com a mensagem informalmente traduzida, o programa completo é:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Oi, gente!" << endl;
}
```

Para executar esse programa e obter na tela do console a saída *Oi, gente!*, o programador precisa antes criar o arquivo com esse texto do código-fonte. Apenas então ele poderá compilar o programa para gerar o arquivo executável correspondente. Essas duas tarefas são detalhadas na seqüência.

Criação do código-fonte

Para o primeiro passo, é importante que o arquivo criado tenha como conteúdo apenas a seqüência de caracteres que compõe o programa. Para tanto, é preciso

utilizar um editor de programas e não um editor de documentos, pois este tipicamente inclui instruções de controle de formatação que, embora não-visíveis para o autor, estão presentes no arquivo. A presença de caracteres estranhos é uma possível causa de erros — o compilador não reconhece os caracteres de controle e, assim, não consegue processar o arquivo.

Para que o compilador possa produzir o arquivo executável, é preciso que o código-fonte esteja de acordo com as regras da linguagem de programação. Se tudo estiver correto, a compilação deve prosseguir até a criação do programa executável. Caso contrário, é papel do compilador indicar da melhor maneira possível em qual ponto do programa ele não conseguiu prosseguir com o trabalho de reconhecimento do código, de modo que o programador possa ter um bom ponto de partida para sanar o defeito.

Obviamente, o compilador não reconhece programas escritos em outras linguagens de programação, mesmo que parecidas. Considere a seguinte versão do mesmo programa, expressa na linguagem Java:

```
class helloJava {
   public static void main(String[] args) {
      System.out.println("Oi, gente!");
   }
}
```

A tentativa de compilar esse programa com um compilador para a linguagem C++ indicaria um erro com uma mensagem do tipo

...: parse error before 'static'

Nesse caso, como as duas linguagens são parecidas e compartilham palavras reservadas, o erro foi na tentativa de obter o sentido das palavras combinadas (*parse error* indica que o erro ocorreu durante a etapa de análise sintática) e só foi indicado na segunda linha do código.

Se o programa fosse em uma linguagem menos similar, a quantidade de erros detectados aumentaria. Considere o código em Pascal:

```
program HelloWorld(output);
begin
  WriteLn('Oi, gente!');
end
```

Se o compilador C++ recebesse esse código, ele indicaria erros por não reconhecer as palavras-chave da linguagem (`program`, `begin`) e também por ter

entre as aspas simples mais de um caractere, o que é inválido em C — aspas simples são usadas para um único caractere e aspas duplas para seqüências de caracteres.

Indicação de erros de compilação

Durante a análise do arquivo de entrada, na construção da estrutura do programa, o compilador pode se deparar com situações que não consegue resolver. É importante que o compilador consiga apresentar, em tais condições, indicações tão precisas quanto possíveis de quais são os defeitos de programação que estão presentes na entrada e que impediram o reconhecimento de uma estrutura de comando válida.

Mesmo que não haja erros relativos à escolha da linguagem de programação ou ao editor utilizado para criar o código-fonte, um pequeno engano de programação é suficiente para impedir a criação do arquivo executável. É tarefa do compilador indicar, da melhor maneira possível, a existência desse engano e fornecer indicações para que o programador possa sanar o problema.

Por exemplo, ao omitir o caractere terminador de comando (`;`) ao final da quarta linha do arquivo original em C++, o compilador emite uma mensagem de erro:

```
...cpp: In function 'int main()':
...cpp:5: parse error before '}' token
```

Observe que a mensagem indica que o problema ocorre na quinta linha, que é quando o compilador detectou a ausência do terminador de comando. De fato, esse terminador poderia estar na linha seguinte, e o programa compilaria corretamente.

Outras vezes, um único engano no código pode gerar muitas mensagens de erro. É o que ocorre quando o compilador não consegue identificar claramente a causa do problema, o que faz com que haja uma propagação de mensagens de erro a partir daquele ponto do código. Por exemplo, quando o programador omite o caractere de abertura da seqüência de caracteres (`"`) da quarta linha, o compilador gera as seguintes mensagens:

```
...cpp: In function 'int main()':
...cpp:4: 'Oi' undeclared (first use this function)
...cpp:4: (Each undeclared identifier is reported
     only once for each function it appears in.)
...cpp:4: 'gente' undeclared (first use this function)
```

```
...cpp:4: parse error before '!' token
...cpp:4:21: warning: multi-line string literals are
     deprecated
...cpp:4:21: missing terminating " character
...cpp:4:21: possible start of unterminated string
     literal
```

Observe que o compilador, ao encontrar os símbolos `Oi` e `gente` na linha 4 do código-fonte, assumiu que estes seriam variáveis do programa e que não haviam sido declaradas. Como todas as variáveis em C++ devem ser declaradas antes do uso, a mensagem indicou esse erro — o compilador não teve como detectar que a causa do problema foi a ausência do caractere " para o início da cadeia de caracteres. Nas três últimas linhas, o compilador indica a presença do caractere " na linha 4, coluna 21, mas assume que este inicia uma seqüência de caracteres e por isso reclama que não encontrou o final correspondente. Cabe ao programador reconhecer que não é esse o problema e assim procurar o defeito em algum ponto anterior à indicação da primeira mensagem de erro. Por esse motivo, quando há mensagens de erro de compilação, a estratégia adequada é resolver os defeitos na ordem em que as mensagens aparecem.

Os exemplos de defeitos citados são verificados pelo compilador durante a fase de análise sintática, pois levam em consideração o relacionamento entre elementos básicos que compõem a estrutura dos comandos. Outro tipo de erro detectado pelo compilador é aquele detectado durante a análise léxica, que reconhece os símbolos que representam os elementos dos comandos de forma isolada. Por exemplo, a compilação de um programa com uma declaração como:

```
int 1var;
```

apresenta a mensagem de erro associada à má formação do nome da variável — neste caso, o compilador, ao identificar como primeiro caractere do símbolo um dígito, assumiu que o programador estaria escrevendo uma constante numérica:

```
...: invalid suffix on integer constant
```

O programa compilador deve conhecer todas as regras associadas à formação de um programa correto na sua linguagem-alvo e, dado um código-fonte nessa linguagem, verificar se essas regras estão sendo obedecidas. As regras que indicam como deve ser um programa correto são expressas por meio de gramáticas. A etapa de análise de um programa verifica a aplicação dessas regras, e

apenas para programas cuja análise tenha sido corretamente concluída é possível concluir com sucesso a compilação, com a síntese do programa equivalente em outro formato.

1.4.2 Exemplo de processamento XML

A estrutura básica de arquivos XML foi apresentada na Seção 1.2.1, na qual foi também apresentado o paralelo que existe entre a compilação de um programa e a interpretação de um arquivo nesse formato.

Para ilustrar o que ocorre no processamento de um arquivo XML, um exemplo muito simples será utilizado, a partir de uma hipotética aplicação de controle de registros de uma biblioteca:

```
<livros>
  <livro>
    <título>Dom Casmurro</título>
    <autor>Machado de Assis</autor>
    <ano>1900</ano>
  </livro>
  <livro>
    <título>As Intermitências da Morte</título>
    <autor>José Saramago</autor>
    <ano>2005</ano>
  </livro>
</livros>
```

Esse arquivo, `livros.xml`, contém um elemento raiz (`livros`) que contém a descrição de dois livros, cada um representado por um elemento `livro`. Para cada livro, as informações são representadas como os conteúdos dos elementos `título`, `autor` e `ano`. É um arquivo bem formado.

O processamento do arquivo pode ser feito com qualquer analisador XML. Nesse exemplo, o analisador Xerces-C é utilizado. Um aplicativo que é distribuído juntamente com esse analisador, para ilustrar sua utilização, é `SAXCount`, que realiza uma varredura do arquivo XML e conta os elementos que ele define, se o arquivo estiver correto. Caso contrário, o analisador gera uma mensagem de erro.

A execução desse aplicativo para o exemplo gera a resposta:

```
livros.xml: 1 ms (9 elems, 0 attrs, 0 spaces, 118 chars)
```

A resposta mostra que o arquivo tem nove elementos (um elemento `livros` e dois elementos `livro`, `título`, `autor` e `ano`) e 118 caracteres nos conteúdos.

Para mostrar como o analisador sinaliza defeitos no arquivo, considere que a primeira marcação `livro` foi escrita, por engano, `lirvo`. A mensagem apresentada pelo analisador indicaria

```
Fatal Error at file livros.xml, line 6, char 5
  Message: Expected end of tag 'lirvo'
```

Nesse caso, a indicação foi de que a marcação de final de elemento encontrada na linha 6 não corresponde ao elemento que estava aberto. Indicação similar ocorreria se o erro de digitação houvesse ocorrido na marcação de final de elemento em vez de na marcação de início ou quando as marcações não estão devidamente aninhadas na definição dos elementos.

Outra situação que seria detectada pelo analisador é a omissão de uma marcação de final de elemento. Considere que a marcação de final do primeiro livro fosse omitida do arquivo por distração do programador. O analisador indicaria essa situação com a mensagem:

```
Fatal Error at file livros.xml, line 11, char 8
  Message: Unterminated end tag, 'livro'
```

Nesse caso, a indicação do erro está na última linha do arquivo, quando o analisador encontra a marcação de final do elemento raiz.

Quando o documento não tem um elemento raiz, o analisador deve sinalizar a condição de erro. Se as marcações de início e de final do elemento `livros` forem omitidas, o analisador emitirá a mensagem:

```
Fatal Error at file livros.xml, line 6, char 3
  Message: Expected comment or processing instruction
```

A mensagem sinaliza que, na linha 6, na qual o segundo elemento `livro` começa, não poderia haver um elemento — apenas um comentário (em arquivos XML, entre `<!--` e `-->`) ou uma instrução de processamento.

Para encerrar esses exemplos, outra situação que pode ocorrer é que o arquivo termine sem que o analisador tenha conseguido completar a estrutura interna do documento. Se a marcação de final do elemento raiz for omitida, o analisador emitirá a mensagem:

```
Fatal Error at file livros.xml, line 13, char 1
  Message: The input ended before all started tags
           were ended.  Last tag started was 'livros'
```

1.5 Notas e sugestões de leitura

A linguagem de programação C foi desenvolvida por volta de 1972 nos Laboratórios AT&T Bell, nos Estados Unidos. A motivação para que o autor de C, Dennis Ritchie, criasse uma nova linguagem de programação foi a necessidade de ter uma linguagem que fosse portátil, independente da linguagem simbólica de cada processador e que, ao mesmo tempo, gerasse código eficiente e fosse apropriada para o desenvolvimento das ferramentas do sistema operacional Unix. C é uma ferramenta de programação tão básica que praticamente todas as ferramentas suportadas por Unix e o próprio sistema operacional foram desenvolvidas em C [Kernighan and Ritchie, 1986].

C++ foi criada em 1979 por Bjarne Stroustrup [Stroustrup, 2001], também dos Laboratórios Bell, para estender a linguagem C com construções da programação orientada a objetos. Inicialmente denominada *C with classes*, só recebeu o nome C++ a partir de 1983. Na Web, o site *C/C++ Reference* (`http://www.cppreference.com/`) descreve as principais classes do núcleo da linguagem, incluídas as classes para manipulação de streams usadas neste capítulo.

A coleção de compiladores GCC, que inclui compiladores para C e C++, entre outras linguagens, é descrita na página `http://gcc.gnu.org/`. Esses compiladores fazem parte da distribuição padrão dos sistemas Linux.

A linguagem XML foi criada pelo Consórcio W3C (`http://www.w3.org/XML/`), que recomenda a sua utilização para o desenvolvimento de praticamente todas as aplicações para a World Wide Web.

O analisador Xerces-C faz parte da plataforma de aplicações associadas ao servidor Web Apache (`http://xerces.apache.org/xerces-c/`). Além da versão C++ utilizada neste capítulo, o mesmo projeto disponibiliza analisadores XML para outras linguagens de programação.

1.6 Exercícios

1.1 Quantos bits são necessários para representar instruções em código de máquina para os seguintes processadores? Assuma que todos os códigos de operação têm o mesmo número de bits.

   (a) Um processador com 38 instruções que podem ter referências a dois endereços de memória de 32 bits cada um;

(b) Um processador com 32 instruções que podem ter referências a três registradores, sendo que há 16 registradores no processador;

(c) Um processador com 142 instruções que podem ter referências a um endereço de 32 bits.

1.2 Um dos projetistas do processador hipotético da Seção 1.1.1 sugeriu que, em vez das instruções `LOAD` e `STORE`, uma única instrução `MOVE` deveria ser utilizada; nesse caso, a ordem dos operandos serviria para diferenciar a direção da transferência. Por exemplo, `MOVE 1,R0` indicaria a transferência do conteúdo da posição 1 de memória para o registrador R0. Já a instrução `MOVE R0,4` indicaria a transferência do conteúdo do registrador R0 para a posição de memória 4. Por que os projetistas teriam mantido duas instruções separadas?

1.3 Qual é o código binário para as seguintes instruções do processador hipotético da Seção 1.1.1?

(a) `LOAD 1,R0`

(b) `STORE R0,4`

(c) `BZERO R2,15`

(d) `ADD R0,R2,R0`

1.4 Qual é a seqüência de instruções simbólicas correspondentes ao seguintes códigos de máquina do processador hipotético da Seção 1.1.1?

(a) `00000000 10001010 11101010`

(b) `11111111 10101010 01010101 00011011`

1.5 Os projetistas da segunda geração do processador hipotético da Seção 1.1.1 devem considerar as seguintes demandas:

(a) Acrescentar duas novas operações;

(b) Dobrar o número de registradores de dados;

(c) Dobrar a largura dos dados de 8 para 16 bits;

(d) Ampliar a capacidade de endereçamento de 16 para 256 posições.

Qual o impacto isolado de cada uma dessas modificações no formato binário das instruções do processador? E, se todas as modificações forem implantadas, qual será o novo formato da instrução?

1.6 Um programa em C ou C++ permite a passagem de argumentos da linha de comando por meio de dois parâmetros da função `main`:

```
int main(int argc, char *argv[]) {
     ...
}
```

O primeiro parâmetro, que tipicamente recebe o nome `argc` (*argument count*), indica o número de palavras (separadas por espaços) presentes na linha de comando, incluindo o próprio nome do programa. O segundo parâmetro, cujo nome típico é `argv` (*argument value*), é um arranjo de ponteiros para caracteres, onde cada elemento do arranjo representa uma das palavras da linha de comando. Com o uso desses argumentos, desenvolva um programa em C++ para apresentar na saída padrão o conteúdo de um arquivo cujo nome é fornecido na linha de comando.

1.7 Com o uso de `argc` e `argv`, definidos anteriormente, desenvolva um programa em C++ para implementar a cópia do conteúdo de um arquivo, cujo nome é passado como o primeiro argumento para o programa na linha de comando, para outro arquivo, cujo nome é passado como o segundo argumento na linha de comando.

1.8 Com o uso de `argc` e `argv`, definidos anteriormente, desenvolva um programa em C++ para contar o número de caracteres, palavras e linhas no arquivo cujo nome foi especificado na linha de comando e apresentar esses totais na tela (saída padrão).

1.9 Qual é o erro associado a cada uma das seguintes declarações de variáveis em um programa C++? Com o auxílio de um compilador C++, interprete as mensagens associadas a esses erros.

   (a) int do;

   (b) int valor = 078;

   (c) char a.c = 0;

   (d) char b = 715.

1.10 A função `atoi`, da biblioteca padrão da linguagem C, permite a conversão de uma seqüência de caracteres (passada como argumento da função) para um valor inteiro (seu valor de retorno). Use essa função para implementar

uma função C++ que receba qualquer quantidade de inteiros na linha de comando e apresente na saída padrão a soma desses valores. Por exemplo, se o programa executável tem o nome de `total`, a execução

```
total 1 20 100
```

deve apresentar na tela o valor 121.

1.11 A partir do exemplo da Seção 1.4.2, apresente três outras situações de má formação de arquivos XML.

1.12 Por que, no segundo exemplo de erro no processamento do arquivo XML apresentado na Seção 1.4.2, a indicação de ausência da marcação de fim do elemento `livro` só apareceu na última linha do arquivo, quando o analisador encontrou a marcação de final do elemento raiz, e não quando o segundo elemento `livro` foi iniciado?

Capítulo 2

Representações de linguagens

Como qualquer linguagem usada para a comunicação, uma linguagem de programação precisa determinar seu conjunto de elementos básicos (um vocabulário) e como compor sentenças válidas usando esses elementos — ou seja, quais são as regras para determinar que uma sentença esteja sintaticamente correta.

Há mecanismos relativamente simples e formais, especificados por meio de gramáticas, para definir o conjunto de símbolos válidos para a linguagem e o seu conjunto de regras sintáticas. O formalismo usualmente utilizado é a Teoria de Conjuntos, cuja notação é revista na Seção 2.1.

Com esse formalismo, é possível expressar linguagens e seus símbolos (ou alfabetos) e, por meio de padrões de substituição de símbolos, derivar seqüências de símbolos que pertencem à linguagem. A definição formal da linguagem é denominada gramática. Linguagens e gramáticas são os assuntos tratados neste capítulo.

2.1 Notação de conjuntos

A formalização de gramáticas utiliza conceitos elementares da Teoria de Conjuntos. Conjuntos são usualmente representados por seqüências de elementos entre chaves, como em

$$\{1, 2, 3\}$$

Conjuntos podem também ser representados por nomes. Por exemplo, o conjunto dos números naturais é $\mathbb{N}$, ou seja,

$$\mathbb{N} = \{1, 2, 3, 4, 5, \dots\}$$

As reticências (...) indicam que a enumeração dos elementos do conjunto continuaria sem ter fim, ou seja, o conjunto é infinito.

Quando for de interesse atribuir um nome a um conjunto, por convenção são utilizadas letras maiúsculas. Por exemplo, o primeiro conjunto desta seção poderia receber o nome A, ou seja,

$$A = \{1, 2, 3\}$$

O símbolo $\in$ expressa a relação de pertinência, ou seja, é usado para expressar que um dado elemento pertence a um conjunto. Para o conjunto acima, por exemplo, é verdade que

$$1 \in A$$

Para representar que um elemento não pertence a um conjunto, o símbolo $\notin$ é utilizado. Por exemplo, o fato de que 7 não é um elemento do conjunto A pode ser indicado pela expressão

$$7 \notin A$$

O conjunto que não tem nenhum elemento é chamado de conjunto vazio, representado pelo símbolo $\emptyset$ ou pelo conjunto $\{\ \}$. Observe que $\{\emptyset\}$ não representa o conjunto vazio — é um conjunto com um elemento, que é o conjunto vazio.

Os elementos de um conjunto podem ser enumerados, como mostrado, ou podem ser definidos por meio de um predicado, ou seja, pela descrição de propriedades que devem ser verdadeiras para que um elemento faça parte do conjunto. Por exemplo, o conjunto A acima pode ser definido alternativamente como

$$A = \{x \mid x \in \mathbb{N} \wedge x < 4\}$$

A leitura que é feita desta definição é: "São elementos de A todos os valores (x) que façam parte do conjunto de números naturais e (denotado pelo símbolo $\wedge$) que sejam menores que 4." Os únicos valores que satisfazem essa preposição são 1, 2 e 3.

Há diversas operações que têm conjuntos como argumentos e cujos resultados também são conjuntos. Entre essas, as principais são união, interseção e diferença. A operação de união, representada pelo símbolo $\cup$, tem como resultado um conjunto que contém todos os elementos de cada um de seus argumentos. Ou seja, dados dois conjuntos quaisquer B e C,

$$B \cup C = \{x \mid x \in B \vee x \in C\}$$

onde o símbolo $\vee$ representa "ou". Por exemplo, se $B = \{1, 2\}$ e $C = \{2, 3\}$, então

$$B \cup C = \{1, 2\} \cup \{2, 3\} = \{1, 2, 3\}$$

Observe que em conjuntos não há elementos repetidos; assim, o elemento 2 aparece uma única vez no conjunto resultado.

A operação de interseção, representada pelo símbolo $\cap$, tem como resultado um conjunto que contém apenas os elementos que também pertencem aos dois argumentos. Da mesma forma, para quaisquer dois conjuntos B e C,

$$B \cap C = \{x \mid x \in B \wedge x \in C\}$$

Por exemplo, se novamente $B = \{1, 2\}$ e $C = \{2, 3\}$, então

$$B \cap C = \{1, 2\} \cap \{2, 3\} = \{2\}$$

pois 2 é o único elemento que pertence aos dois conjuntos.

Já a operação de diferença, aqui representada por $-$, resulta num conjunto que contém todos os elementos do primeiro conjunto que não pertençam ao segundo conjunto. Para dois conjuntos B e C,

$$B - C = \{x \mid x \in B \wedge x \notin C\}$$

Para as mesmas definições de B e C do exemplo anterior,

$$B - C = \{1\}$$

e

$$C - B = \{3\}$$

Um conjunto B é um subconjunto de outro conjunto C, representado pela relação $B \subseteq C$, se todos os elementos de B também são elementos de C. Se $B \neq C$, então a relação pode ser expressa $B \subset C$. Por exemplo, são relações verdadeiras:

$$\{1, 2, 3\} \subseteq \{1, 2, 3\}$$
$$\{1, 2\} \subset \{1, 2, 3\}$$

Uma partição de um conjunto D é um conjunto de subconjuntos $D_1 \subseteq D$ e $D_2 \subseteq D$ tal que

$$D_1 \cup D_2 = D$$
$$D_1 \cap D_2 = \emptyset$$

ou seja, todos os elementos de D pertencem a algum subconjunto e nenhum elemento de D pertence a mais de um subconjunto.

2.2 Linguagens

Toda linguagem — natural ou artificial — é definida a partir da combinação de um conjunto de símbolos básicos. Esse conjunto é denominado alfabeto da linguagem. Por exemplo, para a língua portuguesa, as palavras devem ser formadas a partir de combinações de símbolos do alfabeto {a, b, c ..., z, A, ..., Z}. Já para o código de máquina de um computador, as palavras válidas devem ser formadas pela combinação de símbolos do alfabeto $B = \{0, 1\}$.

Qualquer combinação (ou concatenação) de símbolos de um alfabeto é denominada *string* naquele alfabeto. Por exemplo, se uma linguagem é definida como "qualquer string do alfabeto binário B", então 0, 001 e 1110101 são strings válidas para essa linguagem, mas 012 ou $a0b$ não o são.

Assim como na notação de conjuntos é preciso representar, em algumas situações, o conjunto que não tem nenhum elemento, também há situações nas quais é necessário ter uma representação para uma string sem nenhum símbolo, denominada string vazia. Uma representação usual para a string vazia é o símbolo ε.

Se o conjunto A é um alfabeto, então a clausura de A, denotada $A*$, é o conjunto de todas as strings — incluindo a string vazia — compostas a partir de símbolos de A. Essa operação também é conhecida como estrela de Kleene, em homenagem ao matemático que propôs sua definição. Por exemplo, para os símbolos em representação binária, a clausura de B é

$$B* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \dots\}$$

A notação A^+ representa o conjunto de todas as strings compostas a partir do alfabeto A sem incluir a string vazia, ou seja, o conjunto de strings do alfabeto A com pelo menos um símbolo. Na notação de conjuntos, isso é representado pela expressão

$$A^+ = A * -\varepsilon$$

Nem todos os símbolos que pertencem à clausura de um alfabeto são símbolos válidos no contexto de uma linguagem. Para um processador, por exemplo, pode haver seqüências de bits que não representam nenhuma instrução válida. Da mesma forma, há combinações de letras que não têm significado em português. Portanto, para definir uma linguagem, é preciso especificar quais são, entre todas as possíveis combinações presentes na clausura de um conjunto de símbolos, as strings válidas ou aceitáveis.

Linguagens simples podem ser definidas diretamente em termos de notação de conjuntos. Considere a linguagem L definida sobre o alfabeto B:

$$L = \{0^n 1^n \mid n \geq 0\}$$

onde b^n representa uma seqüência de n ocorrências do símbolo b. São strings válidas nessa linguagem quaisquer seqüências que iniciem com uma quantidade qualquer de ocorrências do elemento 0 e terminem com a mesma quantidade de ocorrências do elemento 1. Por exemplo, 01, 0011 e 00001111 são exemplos de strings válidas em L, mas 10 ou 00111, não. Como $n = 0$ é um valor possível na definição da linguagem, então a string ε também é uma string válida.

A possibilidade de expressar, de forma genérica, o que é válido ou não é válido em uma linguagem é um dos conceitos essenciais na definição de um compilador. É desse modo que um compilador C++, por exemplo, consegue determinar que a string `1var` não é um inteiro válido ou tampouco um identificador válido e assim emitir uma mensagem de erro quando encontra tal seqüência de símbolos no código. O mesmo princípio é utilizado para identificar que uma declaração `int int;` não é válida.

2.3 Gramáticas

A definição de uma linguagem pela definição direta do conjunto com seus elementos, como acima, é adequada para linguagens simples, cujos elementos podem ser assim enumerados diretamente. No entanto, para linguagens mais complexas, como aquelas envolvidas na programação de computadores, dificilmente essa representação simples é suficiente. Para tanto, é preciso estender esses mecanismos de representação. As gramáticas provêem tais facilidades de representação de linguagens.

2.3.1 Produções

Uma gramática para uma linguagem deve incluir, além da especificação do seu alfabeto, um conjunto de produções. A especificação de uma produção é dada por uma relação representada pelo símbolo $\rightarrow$ para indicar que o lado esquerdo da relação pode ser substituído pelo lado direito. Assim, cada produção determina uma regra para transformar uma seqüência de símbolos em outra.

Além dos símbolos que compõem as strings válidas na linguagem, normalmente é necessário incluir símbolos que serão utilizados apenas como interme-

diários no processo de substituição das seqüências de símbolos por meio da aplicação das produções. Esses símbolos intermediários são denominados símbolos não-terminais. Para diferenciar, símbolos que podem fazer parte das strings na linguagem são denominados símbolos terminais.

A gramática também especifica um símbolo não-terminal que deve ser usado como ponto de partida para a aplicação das regras. Fazem parte da linguagem todas as strings que podem ser obtidas por aplicação dessas produções a partir desse símbolo não-terminal inicial, também denominado símbolo sentencial ou ainda axioma.

Considere novamente a linguagem L definida na seção anterior. Para definir a gramática que descreve essa linguagem, é preciso ter um símbolo que não seja elemento de B para servir como o ponto de partida para a aplicação das regras. Seja Z a letra utilizada para representar esse símbolo não-terminal. Como ε é uma string válida em L, uma possível transformação é

$$Z \to \varepsilon$$

É claro que não haveria nenhuma vantagem se todas as strings válidas de L tivessem de ser enumeradas por meio de regras de transformação, como $Z \to 01$, $Z \to 0011$, e assim por diante. O objetivo é criar regras que, de forma genérica, definam essas strings. Nesse caso, a regra pode ser definida a partir da observação de que, para cada símbolo 0 incluído à esquerda de uma string válida em L, um símbolo 1 deve ser incluído à direita, de forma a garantir que o mesmo número de zeros e uns ocorram na string. Genericamente, isso pode ser expresso pela produção

$$Z \to 0Z1$$

Essa é uma produção recursiva, pois contém o mesmo símbolo Z do lado esquerdo e do lado direito.

Pelo que foi apresentado, é possível observar que toda gramática pode ser expressa por quatro elementos:

V_T um conjunto dos símbolos terminais, ou seja, símbolos que podem compor strings válidas da linguagem.

V_N um conjunto dos símbolos não-terminais, ou seja, símbolos que são utilizados como auxiliares na expressão das regras da linguagem.

$\mathbb{P}$ um conjunto de produções expressas na forma $\alpha \to \beta$, com $\alpha \in (V_T \cup V_N)^+$ e $\beta \in (V_T \cup V_N)^*$.

$S \in V_N$, o símbolo sentencial, ponto de partida na criação de qualquer sentença na linguagem.

Assim, qualquer gramática G pode ser formalmente representada por uma quádrupla, ou seja, uma lista com esses quatro elementos:

$$G = (V_T, V_N, \mathbb{P}, S)$$

Deve-se observar que o último elemento da lista é o símbolo S e não um conjunto que contém esse símbolo. Afinal, o símbolo sentencial é único em uma gramática e não é necessário ter um conjunto para representá-lo. Também deve estar claro que um símbolo não pode simultaneamente ser terminal e não-terminal na gramática; portanto, $V_T \cap V_N = \emptyset$.

Considere a definição de uma gramática G_1 para a linguagem L. Nesse caso, os símbolos terminais são 0 e 1; Z é o único símbolo não-terminal. O conjunto das produções contém dois elementos, $Z \rightarrow 0Z1$ e $Z \rightarrow \varepsilon$. O símbolo sentencial é Z, o único elemento de V_N. Portanto, a representação formal dessa gramática é dada pela quádrupla

$$G_1 = (\{0, 1\}, \{Z\}, \{Z \rightarrow 0Z1, Z \rightarrow \varepsilon\}, Z)$$

Uma mesma linguagem pode ser gerada por mais de uma gramática. Quando duas gramáticas geram a mesma linguagem diz-se que elas são equivalentes.

2.3.2 Derivações

Numa gramática, há sempre pelo menos uma produção que tem, do lado esquerdo, apenas o símbolo sentencial. O procedimento para construir uma sentença na linguagem representada por uma gramática deve sempre iniciar com o símbolo sentencial, procurar produções que tenham esse símbolo em seu lado esquerdo e substituí-lo pelo lado direito de uma dessas produções.

A substituição de símbolos que combinam com o lado esquerdo de uma produção pelo lado direito correspondente recebe o nome de derivação, representada pelo símbolo $\Rightarrow$. No exemplo da gramática G_1, há duas derivações válidas a partir do símbolo sentencial Z:

$$Z \Rightarrow \varepsilon$$

e

$$Z \Rightarrow 0Z1$$

Quando o resultado de uma derivação é uma seqüência de símbolos que inclui símbolos não-terminais, essa seqüência é denominada forma sentencial — é o caso do resultado da segunda derivação. Numa forma sentencial, é possível realizar mais derivações, desde que haja produções cujos lados esquerdos combinem com alguma subseqüência dos símbolos que ela contém. Quando a seqüência contém apenas símbolos terminais, ela é denominada sentença.

A partir da forma sentencial $0Z1$, uma nova derivação irá substituir o símbolo Z pelo lado direito de uma produção. Se a produção escolhida for $Z \rightarrow 0Z1$ novamente, então o resultado da derivação será

$$0Z1 \Rightarrow 00Z11$$

Essa forma sentencial ainda permite novas derivações pela substituição do símbolo não-terminal Z. Se a produção $Z \rightarrow \varepsilon$ for utilizada, então o símbolo Z será substituído pela string vazia, ou seja, será eliminado da forma sentencial:

$$00Z11 \Rightarrow 0011$$

O resultado é uma sentença, a partir da qual não é possível obter outras derivações. A seqüência completa de derivações que permitiram construir essa sentença é

$$Z \Rightarrow 0Z1 \Rightarrow 00Z11 \Rightarrow 0011$$

Genericamente, uma forma sentencial δ de uma gramática G é dita derivável de outra forma sentencial γ quando δ pode ser obtida de γ a partir da aplicação de uma ou mais produções de G. O sinal $+$ é colocado sobre o sinal da derivação para indicar a eventual omissão de passos na seqüência de derivações, como em

$$\gamma \stackrel{+}{\Longrightarrow} \delta$$

Assim, no exemplo anterior pode-se afirmar que em G_1 é verdade que $Z \stackrel{+}{\Longrightarrow} 0011$. Quando a forma sentencial δ de uma gramática G é obtida a partir da forma sentencial γ pela aplicação de exatamente uma produção, diz-se que δ é imediatamente derivável de γ.

Pela escolha adequada da produção que será aplicada a cada derivação, qualquer sentença válida da linguagem pode ser obtida a partir do símbolo sentencial. Olhando pelo outro lado, dada uma string em um alfabeto, essa string será uma sentença válida somente se houver, na gramática que descreve a linguagem, uma seqüência de derivações que produza a string a partir do símbolo sentencial. Caso contrário, a sentença não pertence à linguagem. Esse é o princípio do reconhecimento de sentenças, que será detalhado na Seção 4.1.

2.4 Classificação de gramáticas

A gramática G_1 citada tem, em todas as suas produções, apenas um símbolo do lado esquerdo, mas esta não é uma regra geral. Outras gramáticas podem ter qualquer combinação de símbolos terminais e não-terminais do lado esquerdo das produções. Na definição formal de gramática, o lado esquerdo de produções não está restrito a um único símbolo não-terminal, pois em uma produção na forma $\alpha \rightarrow \beta$ temos que $\alpha \in (V_T \cup V_N)^+$ — ou seja, o lado esquerdo tem pelo menos um símbolo do alfabeto, mas pode ter vários.

As gramáticas têm, em função da estrutura das suas produções, possibilidades de expressar linguagens de distintas complexidades. Quanto mais complexa for a linguagem, mais elaborado será o procedimento para reconhecer suas sentenças. Uma classificação de gramáticas (e correspondentes linguagens) foi proposta por Noam Chomsky, em meados da década de 1950, segundo seu poder de expressão.

Por essa hierarquia de Chomsky, a estrutura de gramática mais genérica é aquela sem restrições quanto ao tipo de produções que ela pode conter. Essas gramáticas são chamadas de gramáticas tipo 0 ou gramáticas recursivamente enumeráveis. Gramáticas tipo 1, também chamadas de gramáticas sensíveis ao contexto, são aquelas que têm uma ou mais produções com lado esquerdo com mais de um símbolo e com lado direito com pelo menos a mesma quantidade de símbolos.

Esses dois tipos de gramáticas são de pouco interesse prático para compiladores. Apesar de ter grande poder de expressão, a manipulação automática de sentenças de suas linguagens não é prática. Já as outras duas classes de gramáticas são essenciais à construção de compiladores.

Gramáticas tipo 2, ou gramáticas livres de contexto, são aquelas que têm todas as suas produções com apenas um símbolo do lado esquerdo. O lado direito das produções pode ter qualquer quantidade de símbolos e ser recursivo, ou seja, com o mesmo símbolo no lado esquerdo e no lado direito da produção.

Gramáticas tipo 3, denominadas gramáticas regulares, são ainda mais restritas que as gramáticas livres de contexto. Além de terem apenas um símbolo do lado esquerdo das produções, há uma limitação na forma que as produções recursivas podem assumir — elas não podem apresentar a auto-incorporação. Se uma produção de uma gramática G tiver um símbolo não-terminal A que aparece em ambos os lados da produção na forma

$$A \rightarrow \alpha_1 A \alpha_2$$

onde α_1 e α_2 são strings não-vazias, diz-se que a produção tem a propriedade da auto-incorporação. Caso a recursão ocorra apenas no início ou no final do lado direito das produções, a gramática é regular.

Desse modo, é possível classificar a gramática G_1 como uma gramática livre de contexto, pois há uma produção recursiva ($Z \rightarrow 0Z1$) com auto-incorporação, nesse caso com $\alpha_1 = 0$ e $\alpha_2 = 1$.

Considere o seguinte fragmento de gramática definida para representar expressões aritméticas. Seja E o símbolo que representa uma expressão válida. A produção para indicar que uma expressão entre parênteses é também uma expressão válida tem a forma:

$$E \rightarrow (\ E\ )$$

Essa produção também apresenta a auto-incorporação e, portanto, a correspondente gramática também será livre de contexto. Tais tipos de situações são recorrentes em linguagens de programação, onde pares de delimitadores — tais como (e), { e }, [e] em C, assim como **begin** e **end** em Pascal — devem ocorrer de forma balanceada.

Gramáticas regulares, embora tenham poder de expressão mais limitado, são também muito utilizadas para a representação de alguns aspectos importantes de linguagens de programação. Considere a definição de identificadores, que são utilizados para representar nomes de variáveis e de funções. Para que seja válido, um identificador tipicamente precisa ser uma string iniciada por um caractere que é uma letra; após essa letra, a string pode conter outros caracteres que podem ser letras ou dígitos.

Para construir uma gramática que expresse a regra de formação para identificadores, é preciso inicialmente definir símbolos que representem os elementos básicos de formação das strings válidas. De forma simplificada, sejam esses elementos representados pelos símbolos l, para uma letra qualquer do alfabeto, e d, para um dígito qualquer. Esses são os símbolos terminais dessa gramática. Se N é adotado como o símbolo não-terminal que representa um identificador, que no caso dessa gramática é o símbolo sentencial, o seguinte conjunto de produções representa as regras de formação para identificadores válidos:

$$\begin{aligned} N &\rightarrow lR \\ R &\rightarrow lR \\ R &\rightarrow dR \\ R &\rightarrow \varepsilon \end{aligned}$$

A primeira produção expressa que no identificador há pelo menos uma letra no início, seguida de um "resto" R. As demais produções determinam o que é esse resto. Pela segunda produção, ele pode ter uma letra e mais restos; pela terceira, um dígito e mais restos. Como as regras são recursivas, essas duas regras permitem qualquer combinação de letras ou dígitos no identificador. Finalmente, a quarta produção especifica uma expansão não-recursiva para o resto — ou seja, o resto R também pode ser vazio. Como as duas produções recursivas da gramática, a segunda e a terceira, não têm a auto-incorporação, esta é uma gramática regular.

Na construção de compiladores, esses dois tipos de gramáticas são utilizados. Gramáticas regulares são utilizadas para a especificação e o reconhecimento de símbolos isolados; gramáticas livres de contexto, para identificar a estrutura de comandos do programa.

2.5 Notações alternativas

A representação formal de gramáticas, já apresentada, é muito importante para a fundamentação dos princípios que definem e norteiam o reconhecimento automático de sentenças. No entanto, é pouco prática para quem está envolvido na definição de uma linguagem. Para evitar ter de trabalhar exclusivamente com essa notação matemática, notações alternativas para os tipos de gramáticas usadas na construção de linguagens de programação foram propostas e são usualmente utilizadas.

2.5.1 Expressões regulares

Expressões regulares representam uma forma alternativa de expressar uma linguagem regular. Elas são recursivamente definidas pelo seguinte conjunto de regras:

1. A string vazia é uma expressão regular.

2. Dado um alfabeto A, então um elemento de A é uma expressão regular em A.

3. Se P é uma expressão regular em A, então a repetição de 0 ou mais ocorrências de P, denotada $P*$, também é uma expressão regular em A.

4. Se P e Q são expressões regulares em A, então a seqüência de P e Q, denotada PQ, também é uma expressão regular em A.

5. Se P e Q são expressões regulares em A, então a alternativa entre P e Q, denotada $P \mid Q$, também é uma expressão regular em A.

Expressões regulares são criadas a partir da aplicação de três operadores aos símbolos de um alfabeto. Os operadores são concatenação, para denotar a operação de seqüência de símbolos; $|$, para alternativa; e $*$, para repetição. A operação de repetição tem a maior precedência; a alternativa, a menor. Quando necessário, parênteses podem ser usados para alterar ou explicitar essas precedências.

A operação de alternativa é comutativa e associativa, enquanto a concatenação é associativa mas não comutativa. Sejam x, y e z símbolos de um alfabeto. Pela comutatividade, $x \mid y$ é o mesmo que $y \mid x$, mas xy é diferente de yx. Pela associatividade, $x \mid y \mid z$ equivale a $(x \mid y) \mid z$, assim como xyz é o mesmo que $(xy)z$.

Para um dado alfabeto com símbolos a e b, alguns exemplos de expressões regulares são:

$aa*$ strings com pelo menos uma ocorrência do símbolo a, como a, aa e $aaaaa$. Como $*$ tem a maior precedência, essa expressão é diferente de $(aa)^*$, que representa strings com zero ou mais pares de símbolos a.

$a(a|b)*b$ strings que começam com o símbolo a e terminam com o símbolo b e têm qualquer quantidade dos símbolos a ou b no meio. As strings ab, aab, abb, $aabb$ e $abab$ são válidas segundo essa expressão.

$ba|a*b$ a string ba ou uma string com qualquer quantidade (até mesmo 0) do símbolo a terminada pelo símbolo b, como b, ab ou aab.

Observe que para cada expressão regular é possível representar a mesma linguagem por uma gramática regular. A linguagem representada pela expressão $aa*$, por exemplo, pode também ser expressa por uma gramática representada pela quádrupla $(\{a\}, \{S\}, P, S)$ com as seguintes produções em P:

$$S \rightarrow aS$$
$$S \rightarrow a$$

Do mesmo modo, a linguagem representada pela expressão $a(a|b)*b$ pode ser também representada por uma gramática $(\{a, b\}, \{S, Q\}, P_2, S)$ com as seguintes produções em P_2:

$$\begin{aligned} S &\rightarrow aQb \\ Q &\rightarrow aQ \\ Q &\rightarrow Qb \\ Q &\rightarrow \varepsilon \end{aligned}$$

De forma geral, os seguintes padrões podem ser observados:

Concatenação: tem a mesma representação nas duas notações, ou seja, um símbolo após o outro.

Alternativa: o operador | de uma expressão regular corresponde, na gramática, a produções alternativas para um mesmo símbolo não-terminal.

Repetição: o operador de repetição $*$ de uma expressão regular corresponde, na gramática, a uma produção recursiva. Se a expressão permite zero ocorrências do padrão, então a regra que estabelece o fim da recursão leva à string vazia.

Do mesmo modo, é possível obter uma expressão regular a partir de uma gramática regular. Considere o exemplo da linguagem para a especificação de identificadores começados por uma letra. Dadas as definições de l e d da seção anterior, a expressão regular que descreve um identificador é

$$l(l\,|\,d)*$$

ou seja, uma letra seguida por qualquer número (inclusive zero) de letras ou dígitos.

Expressões regulares oferecem uma forma compacta de representar strings de uma linguagem regular. Além disso, elas têm um papel importante na construção de partes de um compilador. O Capítulo 3 apresenta um procedimento sistemático para, a partir de uma expressão regular, construir uma máquina de estados para reconhecer strings da correspondente linguagem regular. Esse procedimento é a base para a construção automática de analisadores léxicos, como o descrito na Seção 3.4.

2.5.2 BNF

A notação BNF (de *Backus-Naur Form*) é uma notação textual, formal, para representar uma gramática livre de contexto. Ela foi inicialmente desenvolvida por John Backus para especificar e documentar a linguagem Algol 60, uma das predecessoras da linguagem C. Peter Naur introduziu posteriormente algumas simplificações que foram incorporadas à notação.

O principal operador dessa notação é o operador binário $::=$, para descrever produções da gramática. O operando do lado esquerdo desse operador é um símbolo não-terminal; o do lado direito é sua expansão, que pode conter símbolos terminais e não-terminais.

Na notação BNF, os símbolos não-terminais são delimitados por colchetes angulares, $<$ e $>$. Quaisquer outros símbolos, exceto aqueles que têm significado especial na notação, são considerados símbolos terminais. Por exemplo, a regra

$$< \text{expr} > ::= \ (< \text{expr} >)$$

define que uma expressão entre parênteses (o lado direito) pode ser substituída por uma expressão (o símbolo não-terminal do lado esquerdo). Nessa produção há um símbolo não-terminal, *expr*, e dois símbolos terminais, (e). Os três símbolos $<$, $>$ e $::=$ são metacaracteres da notação BNF e não fazem parte da linguagem.

A notação BNF também introduz o operador | para expressar, em uma mesma regra, alternativas de expansão. Por exemplo, a regra

$$< S > ::= < A > | < B >$$

é equivalente ao par de regras

$$< S > ::= < A >$$
$$< S > ::= < B >$$

A notação BNF original contém apenas esses operadores. A ampliação do uso da notação BNF fez com que algumas extensões fossem propostas, algumas das quais são comuns nas representações atuais de linguagens. Uma dessas extensões é o operador `[ ]` (opcional), que representa entre os colchetes um conjunto de símbolos que pode ou não ocorrer numa expansão. Por exemplo, a regra

$$< S > ::= \ [< A >] < B >$$

equivale a

$$< S > ::=< A >< B >$$
$$< S > ::=< B >$$

Outro operador que faz parte de extensões usuais da notação BNF é o operador $\star$ (repetição) que, assim como para expressões regulares, expressa que o símbolo que o precede pode ocorrer zero ou mais vezes. Por exemplo, a regra

$$< S > ::= < A >< B > *$$

equivale a

$$< S > ::= < S >< B >$$
$$< S > ::= < A >$$

Uma limitação no uso de BNF é que os símbolos que são utilizados como metacaracteres não podem ser usados diretamente como símbolos da linguagem. Por exemplo, se o operador $*$ é usado como metacaractere para indicar repetição, então não é possível utilizá-lo para indicar a operação de multiplicação numa gramática de expressões aritméticas.

2.5.3 Diagramas sintáticos

Outro tipo de notação usual para gramáticas livres de contexto é a notação de diagramas sintáticos. Nesse tipo de notação, cada símbolo da gramática tem uma representação gráfica; símbolos não-terminais são representados em retângulos, e símbolos terminais, em elipses. A partir dessa representação para os símbolos, as produções para cada símbolo não-terminal são representadas como um grafo dirigido, no qual cada caminho do início ao fim indica a seqüência de símbolos que pode expandir o símbolo representado.

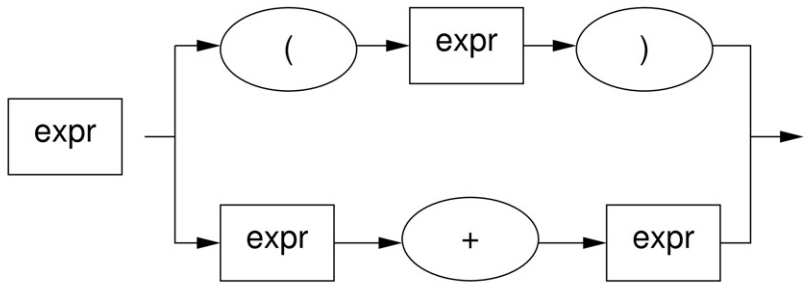
Considere, por exemplo, o seguinte par de expansões para um símbolo não-terminal expressas em BNF:

$$< \text{expr} >::=< \text{expr} > + < \text{expr} > | \ (< \text{expr} >)$$

O correspondente diagrama sintático para essas expansões desse símbolo não-terminal está representado na Figura 2.1.

Nesse diagrama, o retângulo à esquerda que não está conectado ao grafo representa qual é o símbolo não-terminal que é expandido pelo restante do diagrama. Na representação da expansão há dois caminhos possíveis. O caminho

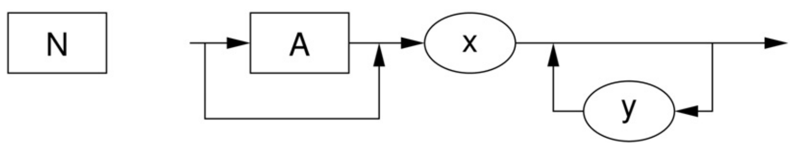
Figura 2.1 Diagrama sintático parcial para uma gramática de expressões



superior é o que representa a expansão pela seqüência de três símbolos: um terminal (abre parênteses), um não-terminal (expr) e outro terminal (fecha parênteses). Outra alternativa é o caminho inferior, que representa a expansão por outra seqüência de três símbolos: o não-terminal expr, o terminal $+$ e novamente o não-terminal expr.

A Figura 2.2 apresenta um outro exemplo de diagrama sintático para uma expressão BNF que inclui as extensões para o operador opcional e de repetição. A ocorrência do operador opcional é representada pelos caminhos alternativos entre dois pontos, um caminho com o símbolo ou símbolos que podem ocorrer e o outro caminho sem nenhum símbolo. Já o operador de repetição é representado no diagrama por um caminho que retorna de um ponto posterior no grafo.

Figura 2.2 Diagrama sintático para a produção $< N >::= [< A >]xy*$



Na representação de uma linguagem, um diagrama principal representa o ponto de partida para as expansões válidas, ou seja, representa as possíveis expansões para o símbolo sentencial. Um diagrama sintático é definido para cada símbolo não-terminal da gramática.

A notação gráfica de diagramas sintáticos tem o mesmo poder de expressão da BNF, porém tem associada uma representação visual para as regras da gramática. Assim, é adequada para fins de documentação e comunicação das

produções da gramática para seres humanos, mas não para processamento por computadores. Como será visto na Seção 4.6, que apresenta um formato para representar gramáticas livres de contexto cuja especificação será processada por um programa, notações ao estilo de BNF são mais apropriadas para essa finalidade.

2.6 Exercícios

2.1 Apresente a lista dos elementos para os seguintes conjuntos:

(a) $C_1 = \{x \mid x \in \mathbb{N} \land x < 7\}$
(b) $C_2 = \{x \mid x \in \mathbb{N} \land 3 < x < 10\}$
(c) $C_3 = C_1 \cup C_2$
(d) $C_4 = C_1 \cap C_2$
(e) $C_5 = C_1 - C_2$
(f) $C_6 = C_2 - C_1$
(g) $C_7 = \{x \mid x \neq x\}$
(h) $C_8 = (C_1 \cup C_2) - (C_1 \cap C_2)$

2.2 Dado o conjunto $A = \{a, b, c, d\}$, responda se as seguintes afirmações são verdadeiras ou falsas:

(a) $a \in A$
(b) $\{a\} \in A$
(c) $a \subset A$
(d) $\{a\} \subset A$

2.3 O conjunto potência de um dado conjunto C é definido como o conjunto que contém todos os possíveis subconjuntos de C. Dado o conjunto $C_1 = \{1, 2, 3\}$,

(a) Qual é o conjunto potência de C_1?
(b) Apresente uma partição para o conjunto potência de C_1 tal que cada partição contenha apenas subconjuntos com o mesmo número de elementos.

2.4 Para o alfabeto binário $B + \{0, 1\}$, apresente exemplos de sentenças para cada uma das seguintes linguagens:

(a) $\{0^n 1^m 0^n | m > 0 \land n \geq 0\}$

(b) $\{1^n 0^{2n} | n > 0\}$

(c) $\{(01)^n 0^n | n \geq 0\}$

2.5 Represente cada uma das linguagens da questão anterior por meio de uma gramática.

2.6 Dada a gramática G_1 (Seção 2.3), mostre que as seguintes formas sentenciais são válidas, ou seja, podem ser obtidas por uma seqüência de derivações a partir do símbolo sentencial:

(a) $000Z111$

(b) 01

(c) 00001111

2.7 Considere a gramática $G_a = (\{a\}, \{S, N, Q, R\}, P, S)$, com o conjunto de produções P com os elementos

$$\begin{aligned} S &\to QNQ \\ QN &\to QR \\ RN &\to NNR \\ RQ &\to NNQ \\ N &\to a \\ Q &\to \varepsilon \end{aligned}$$

(a) Qual é a classificação de G_a pela hierarquia de Chomsky?

(b) Dê quatro exemplos de sentenças que podem ser derivadas a partir do símbolo sentencial S.

(c) A partir de sua resposta para o item anterior, descreva informalmente qual é a linguagem representada por essa gramática.

2.8 Apresente com a notação formal de conjuntos a gramática regular equivalente à expressão regular $(aa)*$. Dê três exemplos de sentenças válidas na correspondente linguagem regular.

2.9 Apresente com a notação formal de conjuntos a gramática regular equivalente à expressão regular $a(b|c)*$. Dê três exemplos de sentenças válidas na correspondente linguagem regular.

2.10 Apresente com a notação formal de conjuntos a gramática regular equivalente à expressão regular $ba|a{*}b$. Dê três exemplos de sentenças válidas na correspondente linguagem regular.

2.11 Apresente com a notação formal de conjuntos a gramática regular equivalente à expressão regular $x{*}(y|z)z{*}$. Dê três exemplos de sentenças válidas na correspondente linguagem regular.

2.12 Apresente, para a seguinte gramática expressa em notação BNF, na qual o símbolo sentencial é `<S>`:

```
<S> ::= ( <M> ) | a | b
<M> ::= <M> ; <N> | <N>
<N> ::= <N> , <S> | <S>
```

(a) A notação formal de conjuntos.

(b) A representação na notação de diagrama sintático.

(c) Três exemplos de sentenças da linguagem descrita pela gramática, com a seqüência de derivações para cada caso.

2.13 Considere a gramática $G_b = \{V_t, V_n, P, S\}$, com $V_t = \{a, b\}$, $V_n = \{A, S\}$ e as produções $P = \{S \rightarrow A,\ A \rightarrow aAb,\ A \rightarrow ab\}$.

(a) Qual é a classificação dessa gramática pela hierarquia de Chomsky?

(b) Represente a gramática em notação BNF.

(c) Represente a gramática em diagramas sintáticos.

(d) Apresente uma seqüência de derivações que resulte na sentença $aabb$.

2.14 Considere a gramática G_c com $V_n = \{S, A, B, C\}$, $V_t = \{x, y, z\}$, símbolo sentencial S e produções $S \rightarrow AxByC$, $A \rightarrow xAx$, $A \rightarrow \varepsilon$, $B \rightarrow By$, $B \rightarrow \varepsilon$, $C \rightarrow zAz$.

(a) Represente a gramática em notação BNF.

(b) Represente a gramática em notação de diagramas sintáticos.

(c) Apresente uma derivação para a sentença $x\,x\,x\,y\,y\,z\,x\,x\,z$.

2.15 Considere a gramática $G_d = (\{x, y, +, \times, (,)\}, \{E\}, P, E)$ onde P é o conjunto com as seguintes produções:

$$E \to E + E$$
$$E \to E \times E$$
$$E \to (E)$$
$$E \to x$$
$$E \to y$$

(a) Classifique a gramática pela hierarquia de Chomsky.

(b) Represente a gramática em notação BNF.

(c) Represente a gramática em notação de diagramas sintáticos.

(d) Apresente duas derivações distintas cujo resultado seja a sentença $x + x \times y$.

2.16 A gramática G_e tem a seguinte descrição na notação BNF:

```
<line> ::= [<line><term>]
<term> ::= <expr>newline
<expr> ::= integer | -<expr>
<expr> ::= <expr>+<expr> | <expr>-<expr>
<expr> ::= <expr>⋆<expr> | <expr>/<expr>
```

(a) Apresente a representação dessa gramática em termos da notação formal de conjuntos.

(b) Apresente a representação dessa gramática em notação de diagramas sintáticos.

(c) Apresente dois exemplos de sentenças produzidas por essa gramática.

2.17 Uma gramática livre de contexto G_f, que tem S como o símbolo sentencial, tem as seguintes produções:

$$S \to aSz \qquad S \to TU$$
$$T \to bT \qquad T \to x$$
$$U \to Ux \qquad U \to b$$

(a) Apresente a representação formal para essa gramática.

(b) Apresente três sentenças na linguagem representada por essa gramática com as correspondentes seqüências de derivações.

Capítulo 3

Análise léxica

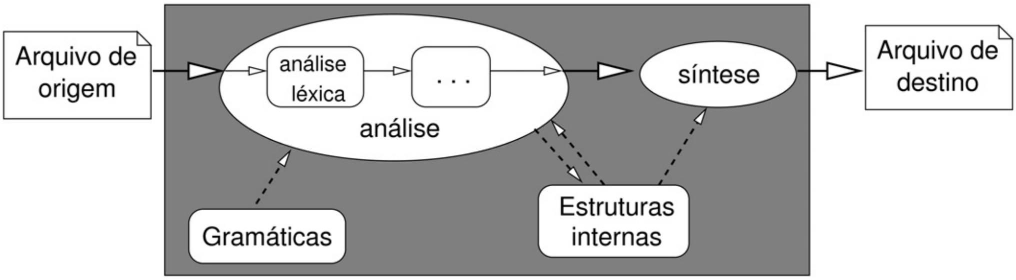
O compilador recebe uma entrada que é uma seqüência de caracteres, tipicamente a partir de um arquivo armazenado em disco. A primeira tarefa que o compilador deve realizar com essa entrada é identificar quais são as palavras (ou, no vocabulário dos compiladores, os tokens) que a compõem e associar, para cada palavra, o seu tipo.

Tal atividade não é muito diferente do que ocorre na interpretação de um texto em linguagem natural. Também nesse caso é necessário primeiro reconhecer as palavras isoladamente e, nesse reconhecimento, identificar sua classe gramatical — quais palavras são substantivos, quais são adjetivos e quais são verbos.

Em razão dessa similaridade, a primeira tarefa que o compilador executa é denominada análise léxica. A diferença é que, no caso de linguagens de programação, as "classes gramaticais", ou os tipos de tokens, são os elementos básicos utilizados na construção de programas — delimitadores, palavras reservadas, identificadores. A Figura 1.4 apresentou um diagrama com as atividades básicas de um compilador; esse diagrama é revisto na Figura 3.1 que destaca, a partir de uma expansão daquela figura, a atividade de análise léxica.

O módulo do compilador que realiza a análise léxica é o analisador léxico. Neste capítulo, os princípios da análise léxica automática são apresentados e sua aplicação na construção de analisadores léxicos de forma automática é ilustrada. Os símbolos que devem ser reconhecidos na análise léxica são representáveis por expressões regulares ou equivalentemente por gramáticas regulares. Existe também uma correspondência unívoca entre expressões (ou gramáticas) regulares e autômatos finitos, máquinas ou programas que podem ser utilizados para reconhecer strings de uma dada linguagem.

Figura 3.1 Atividades do compilador: análise léxica



3.1 Varredura de tokens

No primeiro passo para o reconhecimento de um programa, caracteres são lidos um a um a partir da origem dos dados, que tipicamente é um arquivo em disco. Cada seqüência de caracteres que forma um dos elementos básicos da linguagem deve ser reconhecida como um token. Um procedimento de varredura de tokens deve retornar, a cada invocação, o próximo token reconhecido do arquivo de entrada.

Em uma primeira aproximação, considere um procedimento que reconhece como um token qualquer seqüência de caracteres que esteja separada por um ou mais espaços em branco. Nesse caso, a extração de tokens a partir de um arquivo dá-se da seguinte maneira: o programa analisador recebe a referência para o arquivo aberto e dá início à varredura de caracteres. Cada vez que um caractere de separação de token — espaço, tabulação, mudança de linha — é encontrado, o programa deve deter, em memória principal, uma cópia da string que foi lida e precisa ser reconhecida pelo analisador. A referência para essa string é o retorno do procedimento.

Em um programa C++, a tarefa de ler os caracteres do arquivo pode ser desempenhada com os recursos apresentados na Seção 1.3.1 para a manipulação de arquivos. Para tanto, é preciso ter a referência para o arquivo de origem dos dados — um objeto da classe `ifstream` — que é inicializada ao início do procedimento de reconhecimento e só é liberada ao final.

Em C++, essa necessidade de manter uma informação é resolvida pela definição de uma propriedade de uma classe. Num programa C, a solução equivalente seria manter uma variável global. Se a classe que vai prover o procedimento de extração de tokens for denominada `tokenizer`, então a informação sobre o

arquivo é mantida com um atributo dessa classe:

```
#include <fstream>
class tokenizer {
private:
    ifstream arq;
...
};
```

Além de ler os caracteres, é preciso mantê-los em memória para fornecer o retorno esperado pelo programa que usa o procedimento de varredura de tokens. Essa informação também é mantida como um atributo da classe, na forma de uma variável da classe `string`:

```
#include <string>
class tokenizer {
private:
    string token;
...
};
```

Procedimentos associados a classes são denominados métodos. Nesse caso, a classe deve ter um método de inicialização, para especificar qual é a origem dos dados, e pelo menos um método para retornar o próximo token. Outro método é incluído para indicar se há ou não mais tokens na fonte de dados. Procedimentos de inicialização em C++ são definidos por construtores, métodos com o mesmo nome da classe. A especificação desses métodos da classe pode ser:

```
class tokenizer {
...
public:
    tokenizer(char *nome);
    bool temMaisToken();
    string proximoToken();
};
```

Nesse caso, `nome` deve ser o nome de um arquivo em disco.

Definida a especificação da classe, é preciso definir a implementação desses métodos. A implementação básica do construtor pode ser bem simples — apenas abrir o arquivo indicado:

```
tokenizer::tokenizer(char *nome) {
```

```
    arq.open(nome);
}
```

A implementação básica do método `temMaisToken` também é simples — se o arquivo não estiver no final, então há mais tokens:

```
bool tokenizer::temMaisToken() {
    return !arq.eof();
}
```

Já a implementação do método `proximoToken` deve copiar os caracteres do arquivo para a string até que um dos caracteres de espaço em branco seja encontrado. O operador de entrada » pode ser usado para realizar essa operação, pois ele já incorpora, em sua especificação, o fato de que espaços em branco são ignorados:

```
string tokenizer::proximoToken() {
    if (! arq.eof()) {
        arq >> token;
        return token;
    }
    else return 0;
}
```

Um exemplo de programa que utiliza essa classe para simplesmente apresentar os tokens na saída padrão é:

```
int main(int argc, char** argv) {
    string meuToken;
    tokenizer t(argv[1]);
    while (t.temMaisToken()) {
        meuToken = t.proximoToken();
        cout << meuToken << " : ";
    }
    cout << endl;
}
```

Esse programa assume que o nome do arquivo é passado como argumento na linha de comando. Os tokens extraídos são apresentados numa única linha, separados por dois pontos.

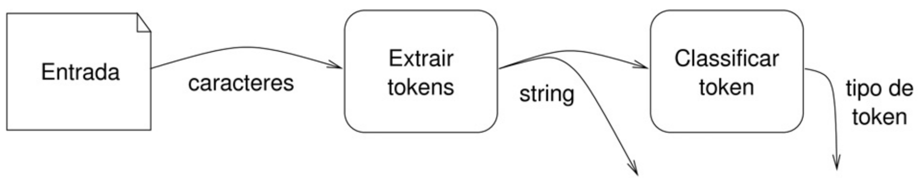
O exemplo de classe aqui apresentado é muito simples e serve apenas para ilustrar o que ocorre durante essa primeira fase do processo de compilação. Caso

seja necessário, uma implementação mais versátil de uma função para separar uma entrada em tokens está disponível na biblioteca padrão de C. É a função `strtok`, que permite indicar outros separadores de tokens para varrer um arranjo de caracteres.

3.2 Classificação de tokens

Obter, do arquivo de entrada, as strings que compõem o programa que será compilado é apenas uma parte da tarefa associada ao analisador léxico. Além de identificar qual é a string que corresponde a um token, o analisador deve também classificar esse token. A estrutura geral do analisador léxico é apresentada na Figura 3.2. Como observado na figura, a saída básica do analisador léxico é um par que combina a string associada ao token e também o seu tipo.

Figura 3.2 Estrutura simplificada de um analisador léxico



Realizar essa classificação do token não é uma tarefa tão simples como ler a string do arquivo. Uma possibilidade seria manter, no analisador, uma tabela que mapeasse strings ao seu tipo. Isso poderia ser aplicado ao reconhecimento de palavras reservadas que correspondem aos comandos de uma linguagem, pois estas são conhecidas e a sua quantidade é limitada. Já para identificadores — nomes de variáveis, classes, funções — há incontáveis combinações.

Na prática, esses dois módulos da Figura 3.2 não precisam ser separados, pois a varredura dos tokens pode ser combinada com a sua classificação. Essa separação apenas destaca essas atividades, como um modo de isolar a tarefa de obter os dados a partir de uma origem da tarefa de classificar uma string.

Para reconhecer esse tipo de símbolo, é preciso ter descrições genéricas e mecanismos que realizem o reconhecimento a partir dessas descrições. A estrutura que permite realizar esse reconhecimento é o autômato finito. O restante desta seção apresenta o procedimento para a construção de um autômato finito a partir da especificação de uma expressão regular.

3.2.1 Autômatos finitos

Um autômato finito é uma máquina de estados finitos que permite reconhecer, por meio de um conjunto de estados e transições dirigidas pela ocorrência de símbolos de um alfabeto, se uma determinada string pertence ou não a uma linguagem regular. Há um estado inicial, que determina o ponto de partida para a realização do reconhecimento da sentença. À medida que caracteres da string de entrada são lidos, o controle da máquina passa de um estado a outro, segundo um conjunto de regras de transição especificadas para o autômato. Se após o último caractere o autômato encontrar-se em um dos estados denominados como finais, a string é reconhecida, ou seja, pertence à linguagem. Caso contrário, a string não pertence à linguagem aceita pelo autômato.

Desse modo, se autômatos finitos forem definidos para as diferentes classes de tokens que podem ocorrer na linguagem, estes podem ser usados como classificadores para realizar a segunda tarefa associada à análise léxica.

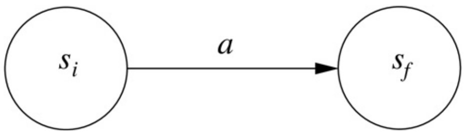
Formalmente, um autômato finito é descrito por cinco características, sendo portanto representável por uma quíntupla $M = (K, \Sigma, \delta, s, F)$:

1. o conjunto finito de estados, K;
2. o alfabeto de entrada finito, Σ;
3. o conjunto de transições, δ, sendo que cada transição é representada por uma tripla (s_i, Σ_T, s_f), em que $s_i \in K$ é o estado de origem da transição, $\Sigma_T \subseteq \Sigma$ é o conjunto de símbolos do alfabeto (caracteres) que disparam essa transição quando o estado corrente é s_i e $s_f \in K$ será o novo estado corrente do autômato após a transição;
4. o estado inicial, s, sendo que $s \in K$;
5. o conjunto de estados finais, F, com $F \subseteq K$.

Autômatos finitos podem ser determinísticos ou não-determinísticos. Um autômato finito não-determinístico pode ter transições na forma (s_i, ε, s_f), ou seja, pode passar de um estado a outro sem a ocorrência de nenhum símbolo na entrada — uma transição pela string vazia. Outra característica que pode estar presente em um autômato não-determinístico é a possibilidade de, a partir de um mesmo estado, ter mais de um destino possível para um mesmo símbolo de entrada. Em outras palavras, se $\alpha \subset \Sigma$, então pode ser que o conjunto de transições δ tenha entre seus elementos (s_i, α, s_f) e (s_i, α, s_g), com $s_f \neq s_g$.

Autômatos finitos são usualmente apresentados na forma de um grafo dirigido, onde estados são representados por círculos e as transições por arestas rotuladas com os símbolos que disparam a transição entre os dois estados conectados. Na Figura 3.3 está representada uma transição que pode ocorrer se o autômato estiver num estado s_i e se o próximo símbolo da string de entrada for a. Nesse caso, após consumir esse símbolo de entrada, o autômato estará no estado s_f. Diz-se nesse caso que a transição de s_i para s_f foi disparada pelo símbolo a.

Figura 3.3 Representação gráfica da transição $(s_i, \{a\}, s_f)$



Ao considerar os elementos da quíntupla que permite representar um autômato finito, a notação gráfica da Figura 3.3 permite representar todos os elementos do conjunto de estados K, que serão todos os estados representados por círculos; o alfabeto de entrada Σ, representado pelos símbolos que rotulam as transições e o conjunto de transições δ, representado pelas arestas entre dois estados. Resta ainda representar o estado inicial e o conjunto de estados finais.

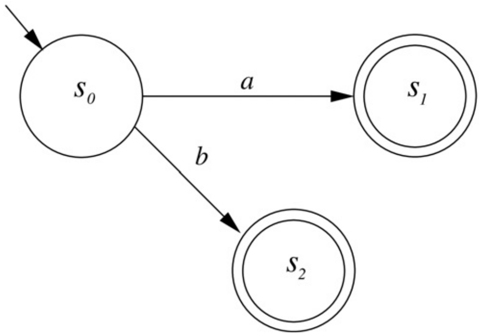
O estado inicial é representado como o único estado no diagrama para o qual há uma transição sem origem e sem rótulo. Estados finais são representados com uma circunferência dupla, em vez da circunferência simples que representa os estados não-finais. Essas duas representações estão ilustradas na Figura 3.4.

A representação formal do autômato finito dessa figura é composta pelos cinco elementos:

Conjunto de estados:	$\{s_0, s_1, s_2\}$
Alfabeto:	$\{a, b\}$
Conjunto de transições:	$\{(s_0, \{a\}, s_1), (s_0, \{b\}, s_1)\}$
Estado inicial:	s_0
Conjunto de estados finais:	$\{s_1, s_2\}$

Uma outra forma de representar as transições de um autômato finito, mais apropriada para fins de processamento automático, é por meio de tabelas de transição. Uma tabela de transição é uma matriz na qual cada coluna está associada a um estado do autômato e cada linha a um símbolo do alfabeto. Cada célula

Figura 3.4 Representação gráfica de estado inicial (s_0) e estados finais (s_1, s_2)



da matriz, correspondente ao cruzamento de uma linha e uma coluna, indica qual é o estado resultante de uma transição a partir do estado indicado na coluna com a ocorrência do símbolo indicado na linha. Assim, a transição expressa na Figura 3.3 poderia ser equivalentemente representada por

	...	s_i	...
a	...	s_f	...
...	...	...	...

Uma célula da tabela de transições pode estar vazia; nesse caso, não há transição possível a partir do estado que está na coluna caso o próximo símbolo da string seja o indicado na linha.

A tabela de transições mantém informação sobre o conjunto de estados do autômato (as colunas da tabela), o alfabeto (símbolos nas linhas da tabela) e transições (combinação de coluna, linha e célula). A informação sobre o estado inicial e os estados finais é mantida em variáveis ou estruturas à parte.

Como exemplo, considere um autômato que recebe uma string binária, composta apenas pelos símbolos 0 e 1, e que deve reconhecer se a string termina com o símbolo 0. Intuitivamente, o autômato pode ser descrito da seguinte forma:

1. A partir de um estado inicial s_i, se ocorrer o símbolo 1, o próximo estado é s_n (para não-aceito), mas se o símbolo for 0, o próximo estado será s_a (aceito).

2. A partir do estado s_n, a ocorrência do símbolo 0 leva o autômato para o estado s_a, mas a ocorrência do símbolo 1 mantém o estado em s_n.

3. A partir do estado s_a, a ocorrência do símbolo 1 leva o autômato para o estado s_n; a ocorrência do símbolo 0 faz com que o autômato permaneça em s_a.

O único estado final desse autômato é s_a, que indica a condição de aceitação da string.

A representação tabular para esse autômato tem três colunas (os estados s_i, s_n e s_a) e duas linhas (os símbolos 0 e 1):

	s_i	s_n	s_a
0	s_a	s_a	s_a
1	s_n	s_n	s_n

Estado inicial: s_i
Estados finais: s_a

Considere como esse autômato processa a string de entrada 010. O estado inicial é s_i e o primeiro símbolo da string é 0; pela informação da tabela (coluna s_i, linha 0), o próximo estado é s_a. O próximo símbolo da entrada é 1. Do estado s_a com símbolo 1, o próximo estado é s_n. O último símbolo da string é 0 que, a partir do estado s_n, leva ao estado s_a. Como não há mais símbolos na entrada, o autômato verifica se o último estado é elemento do conjunto de estados finais. Como é esse o caso, a string é reconhecida como válida.

Já o não-reconhecimento de uma string de entrada pode se dar de duas formas. A primeira é quando o autômato processa todos os símbolos da string e o estado no qual ele terminou não faz parte do conjunto de estados finais. A outra forma ocorre quando, durante o processamento da string, o autômato encontra-se em um estado para o qual não há transição possível com o próximo símbolo.

A estratégia intuitiva aplicada à construção desse autômato, além de inadequada para linguagens um pouco mais complexas, não permite que o processo de construção seja automatizado. Na seqüência, apresenta-se um procedimento sistemático para a construção de um autômato finito para reconhecer strings a partir de uma dada expressão regular.

3.2.2 Construção dos autômatos finitos

A construção sistemática de um autômato finito para reconhecer strings de uma linguagem regular é realizada em três etapas.

A primeira etapa é a construção de um autômato finito que representa diretamente os elementos de uma expressão regular. Pela característica dessa construção, esse primeiro autômato é não-determinístico.

A segunda etapa é a conversão do autômato finito não-determinístico para um autômato finito determinístico equivalente, ou seja, que reconhece strings da mesma linguagem.

A terceira e última etapa tem por objetivo reduzir, se possível, o número de estados do autômato. Para tanto, o procedimento nessa etapa procura identificar estados que sejam redundantes e, se os encontra, os substitui por um único estado.

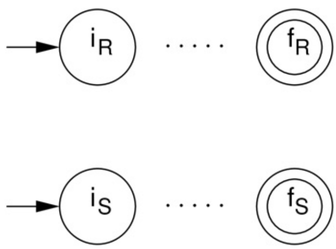
Essas três etapas são descritas na seqüência.

Algoritmo de Thompson

O algoritmo de Thompson define uma seqüência de passos para, a partir de uma expressão regular, como descrita na Seção 2.5.1, obter um autômato finito não-determinístico que reconhece sentenças da correspondente linguagem regular. A construção desse autômato dá-se pela combinação de outros autômatos. Assim como a definição de expressões regulares é recursiva, a construção do autômato por esse algoritmo também o é.

Para a descrição dos passos do algoritmos, considere os dois autômatos finitos A_R e A_S que reconhecem as expressões regulares R e S, respectivamente. Esses autômatos estão representados na Figura 3.5. Todo autômato nesse algoritmo tem um único estado final. A estratégia de construção dos autômatos finitos mais complexos a partir dos autômatos mais simples dá-se por meio de inclusões nesses de transições, sem ocorrência de símbolos (ou seja, pela string vazia), entre estados iniciais e estados finais.

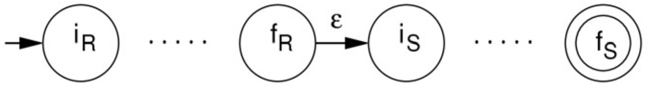
Figura 3.5 Autômatos A_R e A_S



Considere a operação de concatenação. Por definição, se R e S são expressões regulares, então RS também é uma expressão regular. O algoritmo de Thompson descreve uma maneira de construir um autômato finito A_{RS} que reconhece RS a partir dos dois autômatos finitos A_R e A_S. A estratégia é conectar

o estado final de A_R ao estado inicial de A_S com uma transição pela string vazia. Nesse caso, o estado que era final em A_R deixa de ser um estado final, assim como o estado inicial de A_S deixa de ser um estado inicial. O autômato finito resultante é representado na Figura 3.6.

Figura 3.6 Autômato A_{RS}

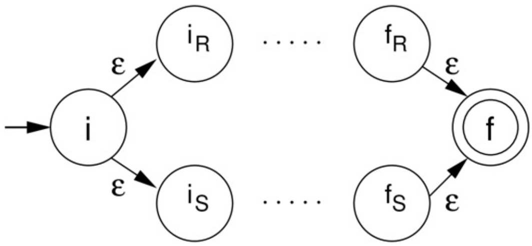


Intuitivamente, é simples entender o raciocínio que leva a essa estratégia de construção. Para reconhecer RS, o autômato finito precisa primeiro reconhecer a expressão R. Por isso, o estado inicial é o mesmo de A_R. Ao final do reconhecimento de R, o autômato precisa ainda reconhecer S. Por esse motivo, o estado final de A_R não é um estado final de A_{RS}, mas deste o autômato deve passar para o estado inicial de A_S. Como não há nenhum símbolo do alfabeto que sinalize a transição de uma expressão para a outra, a transição pela string vazia é usada para construir essa passagem. Depois de reconhecer S, a expressão RS foi reconhecida, motivo pelo qual o estado final de A_S é ainda o estado final de A_{RS}.

De modo similar, um autômato finito $A_{R|S}$ pode ser construído para reconhecer a expressão regular $R|S$ a partir dos dois autômatos que reconhecem R e S. O estado inicial de $A_{R|S}$ é um novo estado, ao qual os estados iniciais de A_R e A_S são conectados por transições com a string vazia. O estado final de $A_{R|S}$ também é um novo estado ao qual os estados finais de A_R e A_S (que deixam de ser finais) são ligados por transições com a string vazia. O autômato finito resultante é apresentado na Figura 3.7.

Também nesse caso é possível analisar intuitivamente como o autômato é construído. O autômato finito $A_{R|S}$ deve reconhecer ou a ocorrência de R ou a ocorrência de S. Portanto, o estado inicial de $A_{R|S}$ não pode ser o estado inicial de A_R nem tampouco o estado inicial de A_S; por esse motivo, um novo estado inicial é introduzido, precedendo esses dois. A partir desse novo estado inicial, o autômato pode reconhecer R — nesse caso, a transição pela string vazia para o antigo estado inicial de A_R é disparada. Na outra alternativa, o autômato pode reconhecer S — por esse motivo, há também a transição pela string vazia do estado inicial para o antigo estado inicial de A_S. Para manter a restrição de que o estado final é único e assim permitir que o autômato resultante possa ser

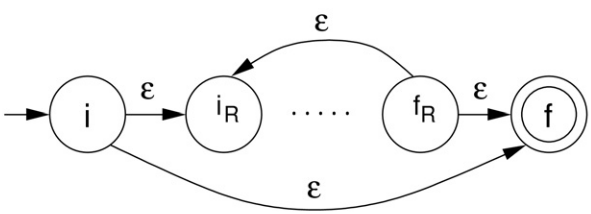
Figura 3.7 Autômato $A_{R|S}$



usado como elemento na construção de outros autômatos, um novo estado final é introduzido. Se a expressão reconhecida foi R, a transição pela string vazia leva do antigo estado final de A_R para esse novo estado final. Se a expressão foi S, então a outra transição pela string vazia leva do antigo estado final de A_S para o estado final de $A_{R|S}$.

A terceira e última forma de composição de autômatos finitos para os operadores de expressões regulares está associada ao operador de repetição. O autômato finito A_{R*} que reconhece a expressão regular $R*$ é construído a partir do autômato A_R. Um novo estado inicial e um novo estado final são introduzidos; esses dois estados são interligados com uma transição pela string vazia, de forma que o autômato possa reconhecer que nenhuma ocorrência da expressão R é uma situação válida. Do novo estado inicial há uma transição pela string vazia para o estado que era inicial em A_R, para permitir o reconhecimento de uma ocorrência da expressão R. Ao fim desse reconhecimento, o antigo estado final de A_R é conectado ao novo estado final por uma transição pela string vazia. Para reconhecer outras ocorrências de R, uma transição pela string vazia é introduzida para conectar o antigo estado final de A_R ao seu antigo estado inicial. O autômato finito resultante é apresentado na Figura 3.8.

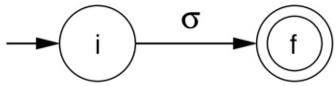
Figura 3.8 Autômato A_{R*}



As três estratégias apresentadas para construir os autômatos finitos associados aos operadores usados na construção de linguagens regulares precisam dos autômatos que reconhecem as expressões regulares que compõem a expressão. Da mesma forma que na definição de uma expressão regular, o elemento não-recursivo na construção desses autômatos está associado ao reconhecimento de um único símbolo do alfabeto.

A base para a construção de expressões regulares afirma que um símbolo do alfabeto é uma expressão regular. O autômato que reconhece um símbolo do alfabeto tem apenas um estado inicial e um estado final, com uma transição do estado inicial para o final rotulado com esse símbolo. Assim, apenas se o símbolo ocorre, o autômato vai para o estado final e a expressão é reconhecida. Se σ representa o símbolo do alfabeto que deve ser reconhecido, então o correspondente autômato A_σ é apresentado na Figura 3.9.

Figura 3.9 Autômato A_σ para reconhecimento de um símbolo σ



A partir da combinação desses elementos e aplicação dessas estratégias, é possível construir o autômato finito não-determinístico para reconhecer qualquer expressão regular.

Na seção anterior, um autômato foi intuitivamente contruído para reconhecer strings compostas pelos símbolos 0 e 1 e que terminassem com uma ocorrência do símbolo 0. A aplicação do algoritmo de Thompson para a construção de um autômato finito não-determinístico para reconhecer essa mesma linguagem tem início pela definição da expressão regular que a representa. Nesse caso, a expressão é

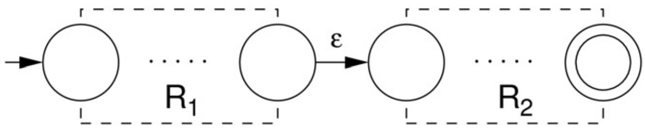
$$(0|1)^*0$$

Os autômatos finitos que serão usados como blocos elementares para a construção são A_0, que reconhece o símbolo 0, e A_1, que reconhece o símbolo 1. Esses dois autômatos são obtidos pela substituição do símbolo σ da Figura 3.9 por 0 e 1, respectivamente.

A combinação desses blocos na construção do autômato para reconhecer a expressão completa é realizada passo a passo, na seqüência em que a expressão regular é construída. A análise da expressão regular deste exemplo leva à seqüência de construção a seguir.

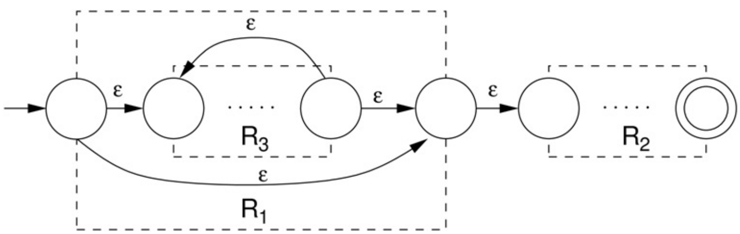
Seja R a expressão regular $(0|1)^*0$. A última operação que é aplicada na construção de R é a concatenação de duas expressões regulares. Sejam essas duas expressões $R_1 = (0|1)^*$ e $R_2 = 0$. Portanto, $R = R_1R_2$ e o autômato que reconhece a expressão R pode ser construído a partir dos dois autômatos que reconhecem R_1 e R_2 pela aplicação da estratégia apresentada na Figura 3.6. O autômato finito para reconhecer R tem assim uma estrutura como a apresentada na Figura 3.10.

Figura 3.10 Autômato para reconhecer a expressão R_1R_2



A expressão R_1, por sua vez, também é construída com um operador e não é uma expressão regular elementar. Ela é a repetição de uma outra expressão regular; seja $R_1 = R_3^*$. Portanto, o autômato que reconhece R_1 é construído a partir do autômato que reconhece R_3 com a estratégia apresentada na Figura 3.8. Desse modo, a estrutura do autômato que reconhece R pode ser detalhada mais um nível, como apresentado na Figura 3.11.

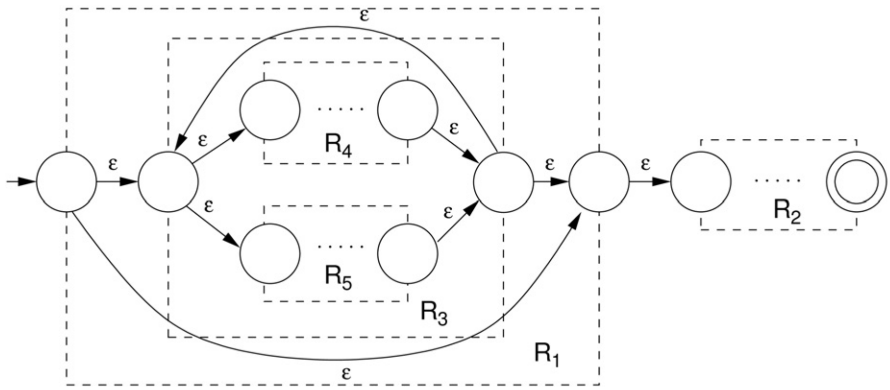
Figura 3.11 Autômato para reconhecer a expressão R_1R_2, com $R_1 = R_3^*$



A expressão R_3 também não é uma expressão regular elementar, pois é a alternativa de duas outras expressões regulares. Seja $R_3 = R_4|R_5$. O autômato que reconhece R_3 pode ser construído com a aplicação da estratégia da Figura 3.7 aos dois autômatos que reconhecem R_4 e R_5. Com essa aplicação, o autômato que reconhece tem a estrutura apresentada na Figura 3.12.

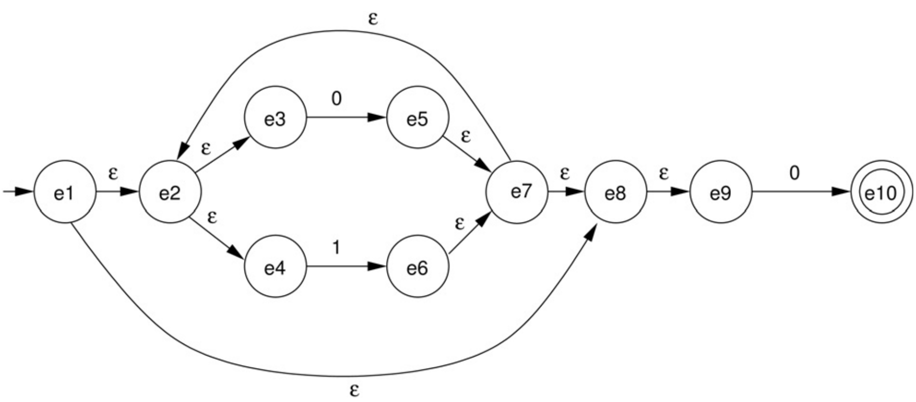
As expressões regulares $R_2 = 0$, $R_4 = 0$ e $R_5 = 1$ estão associadas a símbolos do alfabeto da linguagem e portanto já correspondem aos blocos ele-

Figura 3.12 Autômato para reconhecer a expressão R_1R_2, com $R_1 = R_3^*$ e $R_3 = R_4|R_5$



mentares para a construção do autômato, ou seja, o autômato A_0 para R_2 e R_4 e o autômato A_1 para R_5. Com a substituição desses autômatos na estrutura da Figura 3.12 e a atribuição de rótulos aos estados, o autômato resultante é apresentado na Figura 3.13.

Figura 3.13 Autômato para reconhecer a expressão $(0|1)^*0$



É interessante analisar como esse autômato finito reconhece a string 010. O autômato parte do estado inicial, *e1*, e sem consumir nenhum símbolo passa para o estado *e2* e deste para *e3*. Com o primeiro 0 da string, o autômato faz a transição de *e3* para *e5*. O autômato passa de *e5* para *e7* pela string vazia, assim

como de *e7* para *e2* e de *e2* para *e4*. Com o símbolo 1, realiza a transição de *e4* para *e6*. Com transições pela string vazia passa de *e6* para *e7*, de *e7* para *e8* e de *e8* para *e9*. Com o último símbolo 0, faz a transição de *e9* para *e10*. Como não há mais símbolos e o estado atingido é final, então a string é reconhecida.

Conversão para autômatos determinísticos

Autômatos finitos não-determinísticos não são muito práticos para realizar o reconhecimento de strings. No entanto, como pode ser facilmente observado no exemplo anterior, há diversas situações de ambigüidade. Ao partir do estado *e1*, por que o autômato segue para *e2* e não para *e8*? Por que a transição seguinte foi de *e2* para *e3* e não para *e4*? Sempre que a representação é não-determinística, esse tipo de situação pode surgir.

Para realizar o reconhecimento automático da string de forma mais direta, é mais interessante utilizar um autômato finito determinístico. Nesse caso, para cada combinação de estado corrente e próximo símbolo de entrada existe no máximo uma transição aplicável. Assim, nunca é necessário ter de voltar atrás após uma tomada de decisão por um caminho errado, o que poderia ocorrer se o autômato não-determinístico fosse utilizado diretamente para realizar o reconhecimento.

A boa notícia é que existe um procedimento sistemático para transformar um autômato finito não-determinístico num autômato finito determinístico que aceita a mesma linguagem. O procedimento aqui apresentado é freqüentemente denominado método da construção de subconjuntos. Na descrição a seguir, o termo "estado original" refere-se a um estado do autômato finito não-determinístico obtido com a aplicação do algoritmo de Thompson, enquanto o termo "novo estado" refere-se a estados do autômato finito determinístico resultantes da aplicação do método da construção de subconjuntos.

O princípio associado a esse método é criar novos estados, no autômato finito determinístico, que estejam associados a todas as possibilidades de estados originais em um dado momento da análise da sentença no processo de reconhecimento. Para cada estado original, essas possibilidades incluem o estado do autômato finito não-determinístico e todos os estados que podem ser atingidos a partir dele com transições pela string vazia.

Esse conjunto de estados do autômato não-determinístico é definido pela operação ε^* (lê-se épsilon-clausura). A aplicação da ε^* a um conjunto de estados resulta no conjunto que inclui, além dos próprios estados, cada um dos demais estados do autômato que podem ser alcançados a partir desses com transições pela string vazia.

A aplicação do método da construção de subconjuntos começa pela computação da ε^* do conjunto que contém apenas o estado inicial do autômato finito não-determinístico. O conjunto de estados resultante representa um único estado no novo autômato finito determinístico. Como esse estado inclui o estado inicial do autômato original, será também o estado inicial do novo autômato. Do mesmo modo, se o estado final do autômato original for elemento desse conjunto, o novo estado será também um estado final do novo autômato. Cada novo estado que é gerado dessa maneira é incluído numa lista de estados a analisar.

Considere a aplicação desse passo do método ao autômato finito não-determinístico da Figura 3.13. O estado inicial é *e1*. A $\varepsilon^*\{e1\}$ inclui, além de *e1*, todos os estados que podem ser alcançados a partir de *e1* com a string vazia. Esses estados são *e2*, *e3*, *e4*, *e8* e *e9*. Portanto,

$$\varepsilon^*\{e1\} = \{e1, e2, e3, e4, e8, e9\}$$

Esse conjunto de estados é associado ao estado inicial para o autômato finito determinístico, que pode receber o nome de $s0$.

Na seqüência, é necessário verificar se ainda há estados na lista de estados a analisar. Quando a lista estiver vazia, todas as possibilidades terão sido analisadas e o procedimento estará encerrado. Caso contrário, a análise dos estados deve prosseguir com um estado retirado da lista.

O procedimento prossegue com a análise de estados ainda não analisados. O objetivo dessa análise é verificar, para cada símbolo do alfabeto, se há transição possível a partir do estado sob análise e, se houver transição, para qual estado ela leva. A transição por um símbolo α do alfabeto será possível se houver, no conjunto de estados originais associado ao estado analisado, pelo menos um elemento que tenha a transição pelo mesmo símbolo no autômato original. Se houver, o novo estado resultante da transição será a ε^* do conjunto de estados que resulta da transição por esse símbolo no autômato finito não-determinístico.

Considere a aplicação desse passo do método ao exemplo que foi iniciado anteriormente. No momento, o único estado que existe na lista de estados a analisar é $s0$. Para esse estado, é preciso avaliar se há transição possível com o símbolo 0 e se há transição possível com o símbolo 1. A seguinte tabela sumariza quais são essas possibilidades:

$s0$	$e1$	$e2$	$e3$	$e4$	$e8$	$e9$
0	—	—	$e5$	—	—	$e10$
1	—	—	—	$e6$	—	—

Assim, a partir do estado $s0$, a transição com o símbolo 0 levará a um estado que corresponde a $\varepsilon^*\{e5, e10\}$. A transição com o símbolo 1 a partir do mesmo

estado levará a um estado que corresponde a $\varepsilon^*\{e6\}$. Se os estados resultantes forem novos, devem ser incluídos na lista de estados a analisar. Nesse caso,

$$\varepsilon^*\{e5, e10\} = \{e2, e3, e4, e5, e7, e8, e9, e10\} \qquad (s1)$$

e

$$\varepsilon^*\{e6\} = \{e2, e3, e4, e6, e7, e8, e9\} \qquad (s2)$$

Os dois conjuntos são diferentes do conjunto anteriormente obtido, que foi rotulado como $s0$. Portanto, correspondem a novos estados, que foram rotulados $s1$ e $s2$. Ademais, como $s1$ contém o estado final do autômato original, ele é um estado final para o novo autômato. Esses dois estados são incluídos na lista de estados a analisar. Como a lista não está vazia, o método prossegue com a análise desses estados.

A análise do estado $s1$ resulta em

$s1$	$e2$	$e3$	$e4$	$e5$	$e7$	$e8$	$e9$	$e10$
0	—	$e5$	—	—	—	—	$e10$	—
1	—	—	$e6$	—	—	—	—	—

Portanto, a transição de $s1$ pelo símbolo 0 leva ao estado que corresponde a $\varepsilon^*\{e5, e10\}$, que é o próprio estado $s1$. Como não é um estado novo, a lista de estados a analisar não é alterada. Do mesmo modo, a transição pelo símbolo 1 leva ao estado que corresponde a $\varepsilon^*\{e6\}$, que é o estado $s2$ que já pertence à lista de estados a analisar.

A análise do estado $s2$ resulta em

$s2$	$e2$	$e3$	$e4$	$e6$	$e7$	$e8$	$e9$
0	—	$e5$	—	—	—	—	$e10$
1	—	—	$e6$	—	—	—	—

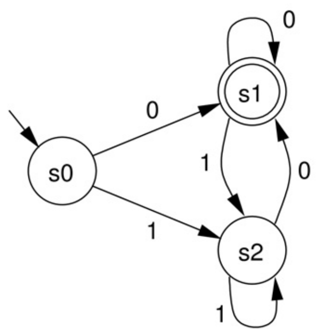
Do mesmo modo, a partir do estado $s2$ a transição pelo símbolo 0 leva ao estado $s1$ e a transição pelo símbolo 1 leva ao estado $s2$. Como nenhum novo estado é gerado, a lista de estados a analisar está vazia e a representação completa do autômato finito determinístico para reconhecer strings da expressão regular $(0|1)^*0$ é obtida:

	$s0$	$s1$	$s2$
0	$s1$	$s1$	$s1$
1	$s2$	$s2$	$s2$

Estado inicial: $s0$
Estados finais: $s1$

Deve-se observar que esse autômato é idêntico ao que foi apresentado na Seção 3.2.1, pois a única diferença está nos rótulos atribuídos aos estados. A representação gráfica para esse autômato é apresentada na Figura 3.14.

Figura 3.14 Autômato finito determinístico para reconhecer a expressão regular $(0|1)^*0$



Pela representação gráfica nem sempre é evidente, mas a representação tabular levanta uma suspeita em relação a esse autômato: se todas as colunas são iguais, é necessário ter mesmo tantos estados diferentes? A resposta a essa questão é o objeto da terceira e última etapa na construção de autômatos finitos para o reconhecimento de strings de uma linguagem regular, que é descrita a seguir.

Minimização de estados do autômato

Em geral, a aplicação do método da construção de subconjuntos produz autômatos com estados redundantes, ou seja, estados que poderiam ser combinados em um único estado sem alterar a linguagem que é reconhecida. Para obter o autômato mais compacto, ou seja, sem estados redundantes, é interessante aplicar um procedimento para a minimização de estados de um autômato.

O procedimento aqui apresentado é baseado na construção iterativa de partições do conjunto K de estados do autômato. A cada iteração, o objetivo é identificar se há um comportamento diferenciado entre os estados que fazem parte de uma partição, ou seja, é preciso analisar o que ocorre nas transições associadas a estados dessa partição. Se há essa diferença de comportamento, então os estados não são redundantes e novas partições são criadas a partir desta. Caso contrário, se todos os estados numa partição têm exatamente o mesmo comportamento, então os estados são redundantes e podem ser combinados, num autômato mi-

nimizado, em um único estado. Se uma partição tem um único estado, então aquele estado não era redundante no autômato original.

O primeiro particionamento do conjunto de estados do autômato reflete a diferença entre estados que são finais e estados que não são finais. Se F representa o conjunto de estados finais do autômato, então essa primeira iteração cria uma partição $P_1 = \{C_1, C_2\}$ com dois subconjuntos, um com os estados finais, $C_1 = F$, e outro com os demais estados, $C_2 = K - F$.

Aplicado ao exemplo do autômato construído para reconhecer a expressão $(0|1)^*0$, esse particionamento inicial é $P_1 = \{C_1, C_2\}$, com

$$C_1 = \{s1\} \qquad\qquad C_2 = \{s0, s2\}$$

Se uma partição P_i contém entre seus elementos um subconjunto não-unitário, então esse subconjunto deve ser analisado para descobrir se seus estados são ou não redundantes. Seja esse subconjunto C_i. A análise é realizada da seguinte forma. Para cada estado de C_i, é preciso verificar para qual subconjunto da partição P_i as transições pelos símbolos do alfabeto levam. Se dois ou mais estados têm transições levando, a partir de cada símbolo, a subconjuntos distintos, então seus comportamentos são distintos e esses estados deverão, na próxima partição P_{i+1}, estar em subconjuntos distintos. Caso contrário, se não houver nenhuma diferença de comportamento entre os estados do subconjunto C_i, então seus elementos são redundantes e podem ser representados por um único estado no autômato minimizado.

No caso do exemplo, a partição inicial P_1 tem dois subconjuntos. Como o subconjunto C_1 é unitário, seu elemento é um estado que não é redundante a nenhum outro estado e deve mesmo ocorrer separado dos demais no autômato minimizado. Portanto, não há nenhum refinamento possível associado a esse subconjunto. O outro subconjunto dessa partição, C_2, tem mais de um elemento e ainda precisa ser analisado.

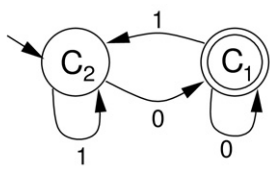
Os elementos do subconjunto C_2 são $s0$ e $s2$. O estado $s0$, na transição pelo símbolo 0, leva ao estado $s1$, que é do subconjunto C_1. O estado $s2$, na mesma transição, também leva a um elemento do mesmo subconjunto. Na transição pelo símbolo 1, ambos os estados também levam a elementos do mesmo subconjunto — nesse caso, de C_2. Ao sumarizar esses resultados, a seguinte tabela é obtida:

	s0	s2
0	C_1	C_1
1	C_2	C_2

Nesse caso, todos os elementos do subconjunto têm exatamente o mesmo comportamento, ou seja, as colunas para $s0$ e $s2$ são idênticas, e esse subconjunto não pode ser mais particionado. Se houvesse outros elementos com pelo menos uma diferença nessas colunas, então uma nova partição teria de ser criada que colocasse cada grupo de elementos com o mesmo comportamento em um mesmo subconjunto e a análise deveria ser repetida para todos os subconjuntos não-unitários.

O refinamento da partição do conjunto de estados deve continuar até que não haja mais possibilidades de particionar nenhum subconjunto da partição, seja porque o subconjunto é redundante, seja porque é unitário. O autômato que resulta desse procedimento de minimização terá um estado associado a cada subconjunto da última partição que foi obtida. No caso do exemplo, como o subconjunto C_2 não pode ser particionado, ele corresponde a um único estado no autômato minimizado. Como não há possíveis refinamentos para essa partição, o autômato resultante, apresentado de forma gráfica na Figura 3.15, tem o número mínimo de estados.

Figura 3.15 Autômato minimizado para reconhecer $(0|1)^*0$



Considere outro exemplo de um autômato finito construído com a aplicação do algoritmo de Thompson e com o método da construção de subconjuntos para reconhecer a expressão regular $a{*}abb{*}$:

	s_0	s_1	s_2	s_3
a	s_1	s_1	–	–
b	–	s_2	s_3	s_3

Para esse autômato finito, s_0 é o estado inicial e $\{s_2, s_3\}$ é o conjunto de estados finais. Na primeira partição, o procedimento separa os estados finais dos demais. Seja essa primeira partição $P_1 = \{C_1, C_2\}$, com $C_1 = \{s_0, s_1\}$ e $C_2 = \{s_2, s_3\}$. A análise do comportamento da partição C_1 do autômato finito para essa primeira iteração resulta em

	s_0	s_1
a	C_1	C_1
b	–	C_2

Como as colunas são diferentes para os estados s_0 e s_1, na próxima iteração esses dois estados estarão em partições distintas. Do mesmo modo, a análise da partição C_2 deve ser realizada:

	s_2	s_3
a	–	–
b	C_2	C_2

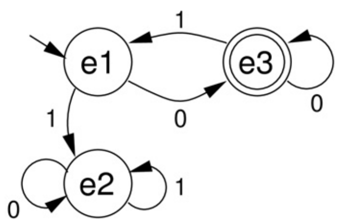
Nesse caso, as duas colunas são iguais, o que indica que os dois estados têm o mesmo comportamento e portanto são redundantes. Assim, esses estados podem ser combinados em um único estado no autômato finito minimizado. Na segunda iteração do procedimento, a partição resultante é

$$P_2 = \{\{s_0\}, \{s_1\}, \{s_2, s_3\}\}$$

Como os estados s_2 e s_3 são redundantes, esta é a partição final.

É possível que o autômato finito resultante da aplicação do procedimento de minimização de estados tenha estados que ainda podem ser eliminados. Uma possível situação na qual isso pode ocorrer é ilustrada na Figura 3.16. Considere o estado $e2$: não é um estado final e não há, a partir dele, nenhuma seqüência de transições que pode levar a um estado final. Este não é um estado redundante, mas é desnecessário para o processo de reconhecimento de strings. Se o autômato chegar a esse estado, a string será rejeitada. O mesmo ocorrerá se o estado e a transição que leva a ele forem eliminados do autômato — apenas a resposta será antecipada.

Figura 3.16 Autômato com "estado morto"



Outra situação similar de “estado morto” que pode surgir no autômato resultante do procedimento de minimização de estados ocorre quando há estados para os quais não há uma seqüência de transições possível desde o estado inicial — ou seja, são estados inalcançáveis. Tais estados mortos podem ser seguramente eliminados, pois essa eliminação não altera a linguagem reconhecida pelo autômato e reduz ainda mais o seu número de estados.

3.3 Analisadores léxicos

A construção do autômato finito que reconhece se uma string pertence ou não a uma dada linguagem foi vista em detalhes na seção anterior. Se a linguagem estiver associada à definição de uma classe de tokens, então não é difícil perceber como a aceitação de uma string por um autômato finito está relacionada à tarefa de classificação de tokens. O próximo passo é traduzir a especificação do autômato finito determinístico num programa de computador, para que essa classificação possa ser feita de forma automática.

Não se pretende aqui entrar no mérito de como desenvolver um programa a partir de uma especificação inicial, que é objeto de estudo da área de Engenharia de Software. Serão abordados apenas alguns aspectos envolvidos na construção do módulo de análise léxica de um compilador.

3.3.1 Visão conceitual

Uma boa forma de se iniciar a construção de um programa é partir de uma descrição do que ele deve fazer. Nesse caso, uma descrição inicial para o analisador pode ser:

> O analisador léxico recebe o nome de um arquivo que contém uma seqüência de caracteres. Para cada token que é extraído desse arquivo, retorna a indicação de se o token é válido e qual é a sua classe. O analisador encerra a execução normalmente após a análise do último token. Caso algum token não seja reconhecido, o analisador indica a situação e passa ao próximo token, se possível; caso contrário, encerra a execução.

O núcleo desse programa é a funcionalidade de reconhecimento de uma string (o token) pertencente a uma linguagem regular. Já tendo visto a fundamentação para o processo de reconhecimento de tokens, é possível definir o

núcleo do analisador léxico como uma implementação de um autômato finito, que reconhece strings como símbolos válidos de uma linguagem ou dá uma indicação de que a string não pertence à linguagem.

A implementação desse analisador léxico requer uma descrição do autômato que reconhece as sentenças da gramática ou expressão regular de interesse. A forma computacional mais simples para representar o autômato é por meio de uma tabela de transições, a partir da qual deve ser possível obter as seguintes informações:

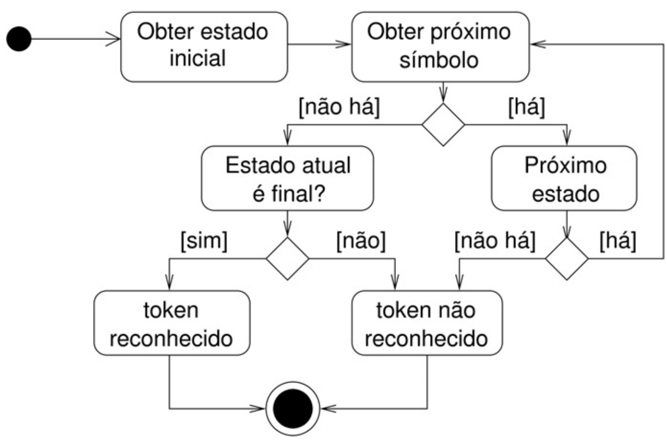
1. O estado inicial para o autômato;

2. Dado um estado qualquer, indicar se este é um estado final (condição de aceitação);

3. Dado um estado qualquer e um símbolo, a indicação de qual é o próximo estado.

O procedimento de reconhecimento é apresentado na Figura 3.17, na forma de um diagrama de atividades na notação UML (*Unified Modeling Language*). A inicialização do procedimento coloca o autômato num estado conhecido, correspondente ao seu estado inicial. A partir desse estado é que os símbolos da string serão analisados.

Figura 3.17 Diagrama de atividades para procedimento de reconhecimento de token



O procedimento prossegue com um laço que tentará analisar cada um dos símbolos da string. O início do laço obtém o próximo símbolo da string, que

inicialmente é o primeiro símbolo. Se não houver próximo símbolo, é preciso verificar se o estado corrente é um estado final. Se for, o procedimento retorna uma indicação de que a string foi reconhecida pelo autômato. Se o estado corrente não for um estado final e não houver mais símbolos na sentença, então deve ser retornada uma indicação de que não houve reconhecimento.

Se houver símbolo a ser analisado, então o procedimento deve continuar o processo de reconhecimento. Para tanto, obtém o próximo estado correspondente à transição do estado atual pelo símbolo sob análise. Se não houver transição possível, então a string não foi reconhecida e o procedimento deve encerrar com essa indicação.

Uma descrição desse procedimento é apresentada no Algoritmo 3.1, que determina se a string σ pertence à linguagem (ou seja, faz parte de uma classe) reconhecida pelo autômato M. O algoritmo utiliza uma variável s para denotar o estado corrente do autômato e uma variável c para o próximo símbolo de entrada da string σ.

Algoritmo 3.1 Algoritmo para um analisador léxico

```
RECONHECE(M, σ)
s ← ESTADO-INICIAL(M)
while TEM-SÍMBOLOS(σ)
do c ← PRÓXIMO-SÍMBOLO(σ)
   if EXISTE-PRÓXIMO-ESTADO(M, s, c)
      then s ← PRÓXIMO-ESTADO(M, s, c)
      else return false
if ESTADO-FINAL(M, s)
  then return true
  else return false
```

Além das variáveis, o algoritmo também faz uso de diversos procedimentos auxiliares. Para o autômato, esses procedimentos são ESTADO-INICIAL, que retorna um estado; ESTADO-FINAL, que retorna verdadeiro se o estado indicado fizer parte do conjunto de estados finais; EXISTE-PRÓXIMO-ESTADO, que retorna verdadeiro de houver transição possível a partir do estado e símbolo indicados; e PRÓXIMO-ESTADO, que retorna o estado decorrente dessa transição. Para a string, o procedimento TEM-SÍMBOLOS retorna verdadeiro se há na string símbolos ainda não processados; o procedimento PRÓXIMO-SÍMBOLO retorna o primeiro caractere da string que ainda não foi processado.

3.3.2 Aspectos de implementação

A implementação do autômato finito por meio de um programa de computador demanda a utilização de algumas estruturas de dados que permitam armazenar a sua representação e, a partir dela, obter as informações necessárias para realizar as operações descritas no algoritmo do analisador léxico. Como apresentado na Seção 3.2.1, são cinco os elementos associados à representação de um autômato finito.

O primeiro elemento é o conjunto de estados. Como pode ser observado nos exemplos trabalhados nas seções anteriores, os rótulos atribuídos aos estados não são relevantes, pois são utilizados apenas para descrever as transições. Alterar esses rótulos não altera o autômato. Assim, qualquer representação simples para os estados pode ser utilizada para representar essa informação. A forma mais simples de representá-los é por meio da atribuição de rótulos inteiros — estado 0, estado 1 e assim por diante. Em C ou C++, valores do tipo `int` podem ser usados para representar estados.

A outra informação básica é o alfabeto de entrada. O autômato reconhece caracteres que são extraídos de um arquivo no formato texto. Desse modo, a representação natural para esses símbolos em C ou C++ é utilizar variáveis do tipo `char`.

As demais informações que representam o autômato finito são derivadas dessas. A mais simples delas pode ser armazenada em uma variável escalar — é a informação sobre qual é o estado inicial do autômato. Como existe apenas um estado inicial para cada autômato, não é preciso usar nenhuma estrutura mais complexa para esse armazenamento. Com essa informação, o procedimento ESTADO-INICIAL pode ser facilmente implementado.

Outra informação que deve ser mantida é o conjunto de estados finais. Embora seja possível manter essa informação como um arranjo, a forma mais simples e eficiente de manter esse tipo de informação em C++ é utilizar os recursos da biblioteca padrão STL (*Standard Template Library*), que oferece um amplo elenco de estruturas de dados e algoritmos parametrizados e que podem ser instanciados de acordo com a necessidade do programador.

Para representar um conjunto de elementos de qualquer tipo, a biblioteca STL oferece a classe parametrizada `set`. Para definir um conjunto com valores inteiros, a seguinte declaração é utilizada:

```
#include <set>
...
set<int> finais;
```

Para adicionar um inteiro ao conjunto, a operação `insert` é utilizada. Por exemplo, se o estado 3 faz parte do conjunto de estados finais, então essa informação pode ser registrada com a operação:

```
finais.insert(3);
```

Dado um valor qualquer, o método `find` é utilizado para saber se ele pertence ao conjunto. Se pertencer, o retorno é um indicador para a posição no qual ele se encontra na coleção — nas definições da biblioteca STL, isso é representado por uma variável do tipo `iterator`, um objeto que permite fazer a verredura nos elementos da coleção. Por exemplo,

```
set<int>::iterator result;
result = finais.find(25);
```

Se o elemento não está presente no conjunto, o valor desse iterador é a posição após o final da coleção, indicado pelo resultado da operação `end`:

```
if (result == finais.end())
  // nao faz parte
```

Para representar as tabelas de transições, é preciso usar uma estrutura mais elaborada. A tabela de transições tem uma estrutura matricial e, portanto, poderia ser representada por um arranjo de duas dimensões. O arranjo é uma estrutura linear de dimensão fixa; uma alternativa para uma estrutura linear cuja dimensão não seja conhecida no momento da programação é o vetor, oferecido na biblioteca STL pela classe parametrizada `vector`. A inserção de um elemento no final do vetor dá-se pela operação `push_back`. Por exemplo, para acrescentar o valor 10 ao final de um vetor de inteiros:

```
#include <vector>
...
vector<int> vint;
vint.push_back(10);
```

No caso da tabela de transições, era possível organizar a informação de modo que cada coluna fosse um vetor, com o estado destino da transição pelo primeiro símbolo na primeira posição (que em C e C++ é a posição 0), o destino da transição pelo segundo símbolo na segunda posição, e assim sucessivamente. Haveria

um vetor desses para cada estado (ou seja, cada coluna da tabela de transição), e esses vetores seriam agrupados num vetor de vetores, no qual o conteúdo da primeira posição seria o vetor com a coluna correspondente ao estado 0, na segunda posição, ao estado 1, e assim por diante.

Apesar de representar corretamente a informação para a tabela de transições, essa abordagem tem um inconveniente: é preciso manter um mapeamento para saber quais caracteres correspondem a quais posições. Uma alternativa que contorna essa limitação é utilizar uma estrutura associativa do tipo mapa para representar as colunas. Um mapa é uma estrutura na qual os elementos armazenados estão organizados na forma de pares *(c, v)*, de tal forma que é possível recuperar o valor *v* (que pode, eventualmente, ser um objeto com estrutura complexa) a partir da chave *c*. Essencialmente, um mapa pode ser visto como uma tabela na qual a posição pode ser de qualquer tipo e não apenas um inteiro. Estruturas desse tipo são denominadas estruturas associativas.

Em C++, a biblioteca STL implementa mapas com a classe `map`. A declaração de uma coleção desse tipo especifica dois parâmetros, o primeiro para o tipo da chave e o segundo para o tipo do valor:

```
#include <map>
...
map<char,int> coluna;
```

Neste exemplo, a chave é uma variável do tipo `char` e o valor associado é um inteiro — ou seja, uma estrutura como se deseja para cada coluna da tabela de transição.

Para operar com os elementos de um mapa, o operador `[]` é sobrecarregado. Por exemplo, para definir que a transição do estado 0 pelo caractere `a` leva ao estado 1 e pelo caractere `b` leva ao estado 2, as seguintes operações seriam realizadas:

```
    coluna['a'] = 1;
    coluna['b'] = 2;
```

A tabela de transições, nesse caso, pode ser um vetor cujos elementos são mapas:

```
vector< map<char,int> > tabTrans;
...
tabTrans.push_back(coluna);
```

Com essa estrutura, o procedimento para apresentar o próximo estado, dados o estado atual e o próximo símbolo da seqüência, torna-se trivial:

```
int proximoEstado(int estadoAtual, char simbolo) {
    return tabTrans[estadoAtual][simbolo];
}
```

Com esses elementos que fazem parte de uma distribuição padrão da linguagem C++, é possível definir todos os elementos de informação necessários para a construção de um analisador léxico. Na próxima seção serão apresentadas algumas ferramentas da programação de sistemas que automatizam boa parte desse trabalho de implementação.

3.4 Geradores de analisadores léxicos

Claramente, os procedimentos até aqui apresentados definem o ferramental necessário para a construção de um analisador léxico para reconhecer sentenças definidas por uma dada expressão regular. Um autômato correspondente à expressão é construído pela aplicação do algoritmo de Thompson, do método da construção de subconjuntos e pelo procedimento de minimização de estados. Esse autômato pode então ser implementado por um programa que executa o algoritmo descrito.

Embora seja possível implementar analisadores léxicos com a construção do autômato finito para cada expressão regular que define um tipo de token e a correspondente implementação em C++, é fácil perceber quão trabalhosa é essa abordagem para qualquer aplicação não-trivial. Como essa complexidade é freqüente na programação de sistemas, diversas ferramentas de apoio a esse tipo de programação foram desenvolvidas.

Uma classe dessas ferramentas são os geradores de analisadores léxicos, que automatizam o processo de criação do autômato e o processo de reconhecimento de sentenças regulares a partir da especificação das expressões regulares correspondentes. Uma das ferramentas mais tradicionais dessa classe é o programa `lex`, originalmente desenvolvido para o sistema operacional Unix. O objetivo do `lex` é gerar uma rotina para o analisador léxico em C a partir de um arquivo contendo a especificação das expressões regulares. Para cada expressão regular é possível associar trechos de código C do usuário que são executados quando sentenças correspondentes àquelas expressões são reconhecidas.

Atualmente há diversas implementações de `lex` para diferentes sistemas,

assim como ferramentas similares que trabalham com outras linguagens de programação que não C. A descrição a seguir é baseada na ferramenta gratuita `flex`, disponível para diversos sistemas operacionais.

3.4.1 Arquivo de especificação

O ponto de partida para a criação de um analisador léxico com uma ferramenta do tipo de `lex` é criar o arquivo com a especificação das expressões regulares que descrevem os itens léxicos que são aceitos. Esse arquivo é composto por até três seções: definições, regras e código do usuário. Essas seções são separadas pelos símbolos `%%`.

A seção mais importante é a seção de regras, onde são especificadas as expressões regulares válidas e as correspondentes ações do programa. Cada regra é expressa por um par na forma

```
padrao   acao
```

contendo a indicação de um padrão e uma ação correspondente a ser executada quando esse padrão é reconhecido.

Padrões

Para a descrição do padrão, a ferramenta define uma linguagem para descrição de expressões regulares. Essa linguagem preserva a notação para expressões regulares apresentada na Seção 2.5.1, ou seja, a presença de um caractere a indica a ocorrência daquele caractere; se R é uma expressão regular, $R*$ indica a ocorrência dessa expressão zero ou mais vezes; e se S também é uma expressão regular, então RS é a concatenação dessas expressões e $R|S$ indica a ocorrência da expressão R ou da expressão S. Além dessas construções, a linguagem de especificação de `lex` oferece ainda diversas extensões, descritas na seqüência.

Uma das extensões é o significado especial atribuído ao caractere ponto (`.`), que é utilizado para representar qualquer caractere exceto `\n`. Assim, o padrão `a.z` representa qualquer seqüência de três caracteres tal que o primeiro seja `a` e o terceiro seja `z`, como `aaz`, `abz` e `a9z`.

Outra extensão utiliza colchetes para representar uma classe de caracteres alternativos. Por exemplo, o padrão `[xyz]` é utilizado para representar um caractere da classe — `'x'` ou `'y'` ou `'z'`. A classe pode também ser representada por uma faixa contígua de caracteres, com a indicação apenas do primeiro e do último caracteres da faixa separados por hífen. Por exemplo, o padrão `[a-f]`

representa a classe com qualquer caractere entre 'a' e 'f', o que inclui os caracteres 'b', 'c', 'd' e 'e'. É possível ainda usar o caractere ^ para negar uma classe. O padrão [^xyz] representa a classe de caracteres [xyz] negada, ou seja, qualquer caractere exceto 'x' ou 'y' ou 'z'.

Há também extensões para representar de forma compacta repetições de um padrão R. Por exemplo, a especificação de um padrão para denotar uma ou mais ocorrências da expressão regular R é representada na notação original de expressões regulares como R R*; na linguagem de especificação de lex, o padrão R+ tem o mesmo significado. Outra notação compacta é R? para denotar R | ε, ou seja, nenhuma ou uma ocorrência da expressão regular R. Chaves podem ser utilizadas para indicar uma quantidade específica de repetições de um padrão. O padrão R{4} representa exatamente quatro ocorrências da expressão regular R; o padrão R{2,} indica que na seqüência de entrada deve haver pelo menos duas ocorrências da expressão regular R; e o padrão R{2,4} indica que a expressão regular R pode ocorrer duas, três ou quatro vezes.

Finalmente, há extensões para indicar se uma expressão ocorre no início ou no final de uma linha. O padrão ^R representa a situação na qual a expressão regular R ocorre no início de uma linha. O padrão R$ representa a ocorrência da expressão regular R no final de uma linha. Há também um padrão especial, <<EOF>>, para indicar a situação de que o analisador chegou ao fim de arquivo.

Caso se deseje usar um dos caracteres que tem significado especial nessa linguagem como um caractere da expressão, é possível usar a construção \X; por exemplo, \. permite especificar a ocorrência de um ponto na expressão regular. Se o caractere X após a contrabarra é tal que \X tenha significado especial em C, ou seja, se X $\in \{0, a, b, f, n, r, t, v\}$, então o caractere recebe a mesma interpretação associada às definições da linguagem C. Outra forma de indicar que uma string deve ser interpretada literalmente é representá-la entre aspas na regra.

Ações

A ação associada a cada padrão na regra é um bloco de código C definido pelo usuário. Esse código será incorporado ao código do analisador léxico, sendo executado quando a seqüência de caracteres de entrada for reconhecida pelo padrão especificado. Como qualquer bloco em C, se apenas uma linha de código for especificada, então as chaves de início e fim de bloco podem ser omitidas; caso contrário, devem obrigatoriamente estar presentes.

O exemplo a seguir de especificação de regras no padrão lex determina o reconhecimento de constantes numéricas inteiras, segundo o padrão da linguagem C:

```
%%
[1-9][0-9]*       printf("Dec");
0[0-7]*           printf("Oct");
0x[0-9A-Fa-f]+    printf("Hex");
```

Nas linguagens C e C++, a base de representação associada a constantes numéricas é definida pelo início da string. Se a constante for iniciada com um dígito entre 1 e 9, a base 10 será assumida e apenas os caracteres entre 0 e 9 podem aparecer no restante da seqüência, se houver outros dígitos. Se a constante for iniciada com o dígito 0, a base 8 é assumida e os dígitos adicionais devem estar entre 0 e 7. Para representar constantes na base 16, a string deve ser iniciada com o prefixo `0x` e, na seqüência, deve conter os dígitos hexadecimais, que agregam aos decimais os caracteres de A a F.

A primeira linha do arquivo de especificação começa com o separador de seções, ou seja, nesse caso a seção de definições está vazia. Como não há um separador de seções que marque o final da seção de regras, a seção de código do usuário também está vazia.

O analisador léxico gerado por esse arquivo de especificação irá receber como entrada strings que eventualmente irão conter constantes inteiras. Se uma constante inteira for encontrada, ela será substituída na saída pela string correspondente especificada na ação, para cada um dos formatos de constantes reconhecidos. Assim, se a string de entrada for

```
abc 10 def 017 ghi 0xAF0
```

o analisador léxico gerará a string

abc Dec def Oct ghi Hex

Esse exemplo ilustra a utilização da regra padrão. Quando a seqüência de caracteres encontrada não combina com nenhum dos padrões especificados na seção de regras, a seqüência não é interpretada e seus caracteres são devolvidos para a saída. No exemplo anterior, foi o que ocorreu com as seqüências `abc`, `def` e `ghi`. Caso o projetista do analisador léxico deseje tratar de outra forma as strings não reconhecidas, ele deve especificar um padrão `\.*` ao final da seção de regras que será aplicado para reconhecer qualquer string não reconhecida pelos padrões previamente especificados.

Definições

A seção de definições permite criar representações simbólicas para padrões que podem ser posteriormente utilizadas na seção de regras. Isso é útil principalmente quando um mesmo padrão ocorre em várias regras, mas também pode ser utilizado como um meio de documentar e tornar mais claro o que representa um determinado padrão.

Por exemplo, se nessa seção houver a definição

```
DIGIT    [0-9]
```

então o padrão da regra que reconhece constantes decimais poderia ter sido escrito na forma

```
[1-9]{DIGIT}*
```

O nome que é definido nessa seção é referenciado na seção de regras entre chaves. Assim, a ocorrência na seção de regras de `{defin}` é substituída no padrão pela expansão da definição `defin`.

Outro tipo de especificação que pode estar presente na seção de definições são trechos de código C que devem ser incluídos no início do arquivo. Esse código, especificado nessa seção entre os símbolos `%{` e `%}`, normalmente é utilizado para incluir diretrizes para o pré-processador C, como `#define` e `#include`.

3.4.2 Integração com código de aplicação

O resultado da aplicação do programa `lex`, tendo como entrada um arquivo de especificação de expressões regulares e respectivas ações, é a criação de um arquivo-fonte contendo o código C que implementa o correspondente analisador léxico. Ele está associado à rotina de nome `yylex()`, que é invocada pela aplicação para fazer o reconhecimento dos itens léxicos na seqüência de caracteres da entrada.

A rotina `yylex()` não recebe nenhum argumento e pode retornar um valor inteiro, que no processo de análise léxica pode ser associado a um tipo de token. Essa rotina lê os caracteres de entrada de um arquivo especificado pela variável global `yyin` e envia os resultados de sua análise para o arquivo especificado pela variável global `yyout`; essas duas variáveis são ponteiros para `FILE`, para a manipulação de arquivos em C. Adicionalmente, a última string que foi reconhecida pelo analisador léxico é referenciada pela variável global `yytext`, do tipo ponteiro para caracteres.

A definição padrão das variáveis `yyin` e `yyout` associa-as, respectivamente, ao arquivo de entrada padrão (teclado) e ao arquivo de saída padrão (tela do monitor). Essa definição pode ser modificada pela especificação presente na seção de código do usuário do arquivo `lex`.

Se nenhum código for definido nessa seção, o código de aplicação utilizado é o fornecido na biblioteca de rotinas do `lex`, tipicamente algo da forma

```
int main() {
    yylex();
    return 0;
}
```

Para modificar as definições padronizadas, o código C que altera esse comportamento deve estar especificado nessa seção. Por exemplo, para que o analisador léxico que reconhece as constantes inteiras pudesse ter a opção de obter sua entrada de um arquivo especificado na linha de comando, o arquivo de especificação `lex` apresentado a seguir poderia ter sido utilizado.

```
DIGIT [0-9]
%%
[1-9]{DIGIT}*          printf("Dec");
0[0-7]*                printf("Oct");
0x[0-9A-Fa-f]+         printf("Hex");
<<EOF>>                return 0;
%%
int main(int argc, char *argv[]) {
  FILE *f_in;

  if (argc == 2) {
    if (f_in = fopen(argv[1],"r"))
      yyin = f_in;
    else
      perror(argv[0]);
  }
  else
    yyin = stdin;

  yylex();
  return(0);
}
```

Os símbolos `stdin`, `stdout` e `stderr` são os equivalentes em C aos objetos `cin`, `cout` e `cerr` de C++.

3.4.3 Geração da aplicação

Nesta seção ilustra-se a utilização de uma das implementações do programa `lex`, que é o aplicativo `flex`, disponível para diversas plataformas computacionais. A sintaxe dos comandos apresentados corresponde à utilização do aplicativo com o sistema operacional Unix.

Considere como exemplo que a especificação `lex` apresentada na seção anterior foi escrita em um arquivo que recebeu o nome `unsint.l`, onde `.l` é uma extensão padrão para esse tipo de arquivo. Para gerar o analisador léxico, `flex` é invocado recebendo esse arquivo como entrada:

```
> flex  unsint.l
```

A execução desse comando gera um arquivo-fonte em linguagem C de nome `lex.yy.c`, que implementa os procedimentos do analisador léxico. Para gerar o código executável, esse programa deve ser compilado e ligado com a biblioteca `libfl`, que contém os procedimentos internos padrão de `flex`. Para tanto, a chave de compilação `-lfl` é incluída:

```
> gcc -o aliss lex.yy.c -lfl
```

O arquivo executável `aliss` conterá o analisador léxico para inteiros sem sinal. Se invocado sem argumentos, `aliss` irá aguardar a entrada do teclado para proceder à análise das strings; o término da execução será determinado pela entrada do caractere `control-D`. Se for invocado com um argumento na linha de comando, `aliss` irá interpretar esse argumento como o nome de um arquivo que conterá o texto que deve ser analisado, processando-o do início ao fim.

3.4.4 Exemplo de aplicação

Esta seção apresenta um exemplo de uma aplicação que utiliza um analisador gerado por `lex` para reconhecer valores de um arquivo de entrada. O papel do analisador léxico é reconhecer se os tokens do arquivo representam valores inteiros em representação decimal, octal ou hexadecimal (no padrão de representação da linguagem C) ou valores reais (também como em C, com ponto decimal). A aplicação recebe a informação sobre a classificação dos tokens e totaliza a quantidade de ocorrências para cada tipo.

A primeira seção contém declarações e definições. Nesse caso, deve conter as definições dos tipos de tokens que devem ser reconhecidos pela aplicação — decimais (D), octais (O), hexadecimais (H) ou reais em ponto flutuante (F):

```
%{
#define D    300
#define O    301
#define H    302
#define F    303
#define B    304
#define X    399
%}
%%
```

Além dos tipos de tokens a reconhecer, há também definições para brancos e outros caracteres não reconhecidos (B) e para o fim de arquivo (X). Os valores associados a tais definições são arbitrários; por convenção, valores para tipos de tokens da aplicação são maiores que 255, o valor máximo representável por um inteiro de 8 bits e que está associado à representação de caracteres no padrão ASCII.

A segunda seção do arquivo contém os padrões para reconhecimento dos tipos de tokens e ações correspondentes. Neste exemplo, a ação é apenas o retorno com a indicação do tipo de token:

```
%%
[1-9][0-9]*    { return D; }
0[0-7]*        { return O; }
0x[0-9A-Fa-f]+  { return H; }
[0-9]*\.[0-9]+|[0-9]+\.[0-9]* { return F; }
[ \n\t]+       { return B; }
.              { return B; }
<<EOF>>        { return X; }
```

Nesse caso, como a aplicação é simples, seu código está incorporado ao arquivo de especificação do lex, na terceira seção. Esse código contém, na parte inicial, as declarações das variáveis e o tratamento necessário para a abertura de um arquivo cujo nome tenha sido indicado na linha de comando. O núcleo do código é o laço que invoca a rotina do analisador léxico e faz a contagem dos diferentes tipos de tokens reconhecidos:

```
  while ((tipoToken = yylex()) != X)
    switch (tipoToken) {
      case D: ++totalDec;
              break;
      case O: ++totalOct;
```

```
            break;
      case H: ++totalHex;
            break;
      case F: ++totalFlt;
    }
```

Espaços e tokens não reconhecidos (tipo B) são ignorados na contagem. Ao final, o código apresenta os totais obtidos.

Caso a aplicação fosse mais complexa, seu código poderia estar distribuído por outros arquivos. Neste exemplo, a aplicação e a especificação de padrões para a geração do analisador léxico estão no mesmo arquivo:

```
%{
#define D      300
#define O      301
#define H      302
#define F      303
#define B      304
#define X      399
%}
%%
[1-9][0-9]*    { return D; }
0[0-7]*        { return O; }
0x[0-9A-Fa-f]+  { return H; }
[0-9]*\.[0-9]+|[0-9]+\.[0-9]* { return F; }
[ \n\t]+       { return B; }
.              { return B; }
<<EOF>>        { return X; }
%%
int main(int argc, char *argv[]) {
  FILE *f_in;
  int tipoToken;
  int totalDec = 0,
      totalOct = 0,
      totalHex = 0,
      totalFlt = 0;

  if (argc == 2) {
    if (f_in = fopen(argv[1],"r"))
      yyin = f_in;
    else
      perror(argv[0]);
  }
  else
    yyin = stdin;
```

```
  while ((tipoToken = yylex()) != X)
    switch (tipoToken) {
      case D: ++totalDec;
              break;
      case O: ++totalOct;
              break;
      case H: ++totalHex;
              break;
      case F: ++totalFlt;
    }

  printf("Arquivo tem:\n");
  printf("\t %d valores decimais\n", totalDec);
  printf("\t %d valores octais\n", totalOct);
  printf("\t %d valores hexadecimais\n", totalHex);
  printf("\t %d valores reais\n", totalFlt);
}
```

Se esse arquivo tem o nome `classif.l`, então a geração do analisador léxico com o código da aplicação integrado a ele dá-se por meio da execução do programa `flex`:

```
>flex classif.l
```

A execução do gerador de analisador léxico produz o código em linguagem C no arquivo `lex.yy.c`, que deve ser compilado para a geração do código executável:

```
>gcc -o classif lex.yy.c -lfl
```

Neste exemplo, a aplicação contém apenas código C e não C++. Portanto, o compilador `gcc` foi utilizado para gerar o módulo executável `classif`.

Por fim, é possível executar a aplicação. Considere que o arquivo de entrada `teste.txt` contém

```
0717  0xAF 1.2
123 100. abc
.5 500 0
xyz 041 99
3.1415 ??? fim
```

A execução

```
> ./classif teste.txt
```

produz o seguinte resultado:

```
Arquivo tem:
         3 valores decimais
         3 valores octais
         1 valores hexadecimais
         4 valores reais
```

Este exemplo ilustra bem o princípio básico da operação do analisador léxico num compilador. Um módulo externo ao analisador, neste caso o código da aplicação, invoca o analisador léxico para que seja realizada a classificação do próximo elemento da entrada. Esse módulo usa o elemento reconhecido em seu processamento e volta a solicitar a classificação do próximo elemento, até o fim do arquivo analisado. Num compilador, esse módulo externo é o analisador sintático, que é apresentado no próximo capítulo.

3.5 Exercícios

3.1 Apresente a representação na forma de tabela de transições para o autômato finito da Figura 3.4.

3.2 Construa um autômato finito determinístico com um número mínimo de estados para reconhecer sentenças descritas pela expressão $a(ab){*}b$. Utilize os procedimentos formais para obter o autômato finito não-determinístico, convertê-lo para um autômato finito determinístico e minimizar seu número de estados.

3.3 Dada a expressão regular $(xx){*}(y|z)z{*}$

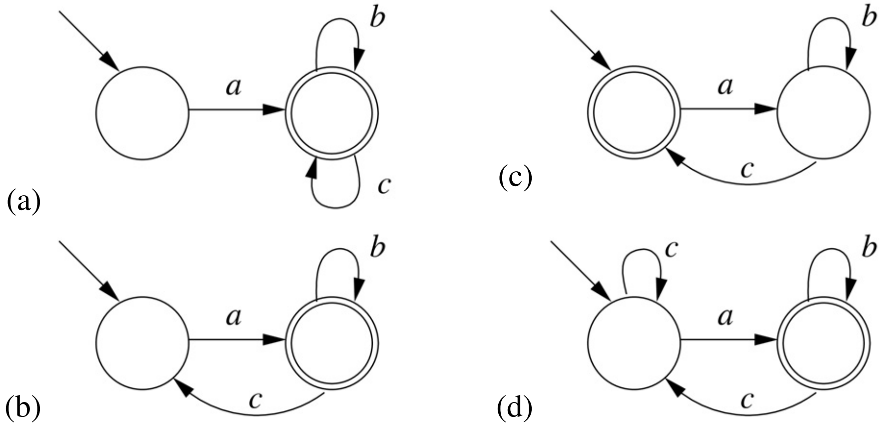
(a) Construa, usando o algoritmo de Thompson, o autômato finito não-determinístico para reconhecer sentenças dessa linguagem.

(b) Converta, usando o método da construção de subconjuntos, o autômato do item *a* para um autômato finito determinístico.

(c) Minimize, se possível, o número de estados do autômato do item *b*.

3.4 Desenvolva o autômato finito determinístico com o menor número de estados para reconhecer sentenças da gramática regular $G = \{V_n, V_t, P, A\}$, com símbolos não-terminais $V_n = \{A, B, C\}$, símbolos terminais $V_t = \{x, y\}$ e produções $P = \{A \to xB, A \to yB, B \to xC, C \to xC, C \to y\}$.

3.5 Apresente as expressões regulares que representam a linguagem reconhecida pelos seguintes autômatos finitos:



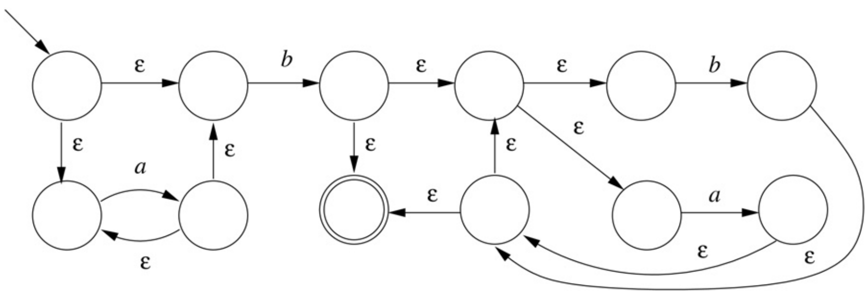
3.6 Em uma aplicação que aceita strings binárias (alfabeto: $\{0,1\}$), as sentenças aceitas são aquelas que terminam com o mesmo bit que iniciaram. Assim, se a expressão inicia com 0, pode ter qualquer combinação de bits (até mesmo vazia) após esse bit desde que haja outro bit 0 no final. Similarmente, sentenças iniciadas com 1 devem terminar com o bit 1.

(a) Apresente uma expressão regular que descreva essas sentenças.

(b) Desenvolva o autômato finito não-determinístico, usando o algoritmo de Thompson, para reconhecer essas sentenças.

(c) Converta o autômato obtido no item anterior para um autômato determinístico com número mínimo de estados.

3.7 A aplicação do algoritmo de Thompson produziu um autômato finito não-determinístico com a seguinte estrutura:



(a) Utilize o método da construção de subconjuntos e o procedimento de minimização de estados para obter um autômato finito determinístico equivalente.

(b) Qual é a expressão regular que descreve as sentenças reconhecidas por esses autômatos?

3.8 Usando a notação de metacaracteres de lex, descreva as expressões regulares que representam as seguintes sentenças:

(a) Todas as seqüências compostas por símbolos 0 e 1 tal que a ocorrência de um 0 é seguida por um ou mais 1's. Observe que a seqüência vazia também é válida.

(b) Uma seqüência que representa um número real, que pode ter um sinal, dígitos na parte inteira, parte fracionária (uma vírgula seguida por dígitos) e expoente (a letra 'e' ou 'E' seguida por um sinal opcional e um ou mais dígitos). Contemple na representação as possíveis combinações usuais para valores reais aceitas em linguagens de programação.

3.9 O seguinte arquivo em formato lex especifica tokens que serão aceitos como entrada em uma aplicação:

```
%%
0[01]{2}0
1(0{2}|1{2})[01]
```

Dada essa especificação, indique quais das seguintes entradas seriam rejeitadas ou aceitas pela aplicação — nesse caso, aponte se pela primeira ou pela segunda regra.

(a) 0000
(b) 0120
(c) 0001
(d) 0010
(e) 0210
(f) 1000
(g) 1010
(h) 1020
(i) 1111
(j) 100110

3.10 A biblioteca padrão de C contém as seguintes funções que realizam a conversão de strings que contêm a representação de valores numéricos para variáveis:

strtol(str, ret, base) Retorna o valor inteiro (tipo `long`) resultante da conversão da seqüência de caracteres iniciada em `char *str`, que deve ser iniciada com brancos ou valores numéricos válidos, de acordo com a `base` indicada. A primeira posição não reconhecida na string é retornada em `char **ret`, que pode ser 0 se esse valor não for utilizado. Caso a base seja 16, a string pode iniciar com o prefixo `0x`.

strtod(str, ret) Retorna o valor real (tipo `double`) resultante da conversão da seqüência de caracteres iniciada em `char *str`, que deve ser iniciada com brancos ou uma representação em ponto flutuante válida. A primeira posição não reconhecida na string é retornada em `char **ret`, que pode ser 0 se esse valor não for utilizado.

(a) Utilize essas funções para estender a aplicação apresentada na Seção 3.4.4 de forma que o total de todos os valores reconhecidos seja também calculado e apresentado pela aplicação.

(b) Modifique as expressões regulares na especificação do analisador léxico para que valores negativos também sejam reconhecidos e computados corretamente no total calculado pela aplicação.

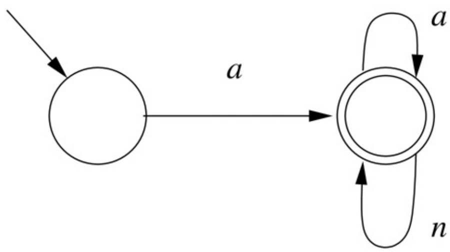
3.11 Apresente uma única expressão regular (a mais compacta possível) que seja capaz de descrever, com o auxílio de metacaracteres da linguagem lex:

(a) Qualquer palavra de 10 caracteres alfabéticos que inicie e termine por vogal;

(b) Qualquer nome de variável de até 5 caracteres alfanuméricos iniciado obrigatoriamente por caractere alfabético;

(c) Qualquer número inteiro, positivo ou negativo, em octal (iniciado por 0), hexadecimal (iniciado por 0x ou 0X) ou decimal (não iniciado por 0). Considere que não há representação para o inteiro zero em decimal.

3.12 Seja o seguinte autômato finito determinístico, capaz de interpretar expressões para uma gramática regular G:



(a) Qual a expressão regular que esse autômato determinístico reconhece?

(b) Faça a tabela de transições de estados para esse autômato.

(c) Escreva seis seqüências de símbolos reconhecíveis pelo autômato.

(d) Com base na expressão regular que você obteve, desenhe o autômato finito não-determinístico correspondente, mostrando as etapas intermediárias em sua construção.

Capítulo 4

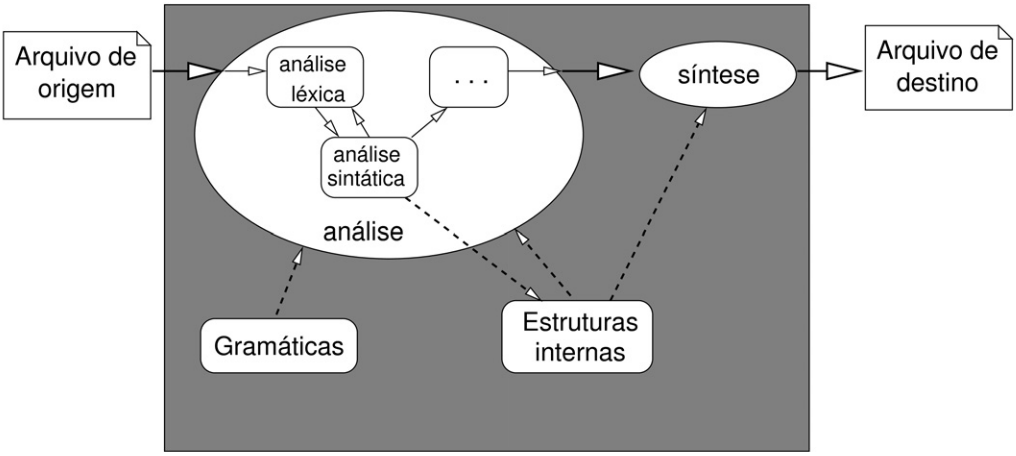
Análise sintática

O Capítulo 3 mostrou o procedimento para reconhecer seqüências de símbolos que satisfazem a especificação por uma gramática regular (tipo 3) usando autômatos finitos e como um correspondente módulo analisador léxico do compilador pode ser implementado. Esse tipo de procedimento é adequado para identificar os símbolos básicos que compõem uma linguagem, mas não para identificar a forma como esses símbolos devem ser combinados. O mesmo ocorre em linguagens naturais: não basta conhecer o significado isolado de cada palavra, é preciso compreender qual é o seu papel na sentença para que a mensagem em uma frase possa fazer sentido. O processo de reconhecer a estrutura de sentenças é a análise sintática, uma das atividades do compilador que é apresentada na Figura 4.1.

Gramáticas tipo 2, ou gramáticas livres de contexto, são adequadas para representar as características de sentenças em linguagens de programação. Embora nem todas as construções de programação sejam passíveis de representação por esse tipo de gramática, com reconhecedores de sentenças livres de contexto e algumas estratégias heurísticas é possível automatizar a análise sintática. O uso e a construção de programas analisadores sintáticos, ou *parsers*, são os objetos deste capítulo.

Tradicionalmente, gramáticas livres de contexto têm sido utilizadas para realizar a análise sintática de linguagens de programação. Nem sempre é possível representar nesse tipo de gramática restrições necessárias a algumas linguagens — por exemplo, exigir que todas as variáveis estejam declaradas antes de seu uso ou verificar se os tipos envolvidos em uma expressão são compatíveis. Entretanto, há mecanismos que podem ser incorporados às ações durante a análise

Figura 4.1 Atividades do compilador: análise sintática



— por exemplo, interações com tabelas de símbolos — que permitem complementar a funcionalidade da análise sintática.

Embora conceitualmente seja possível um arquivo de texto ser completamente transformado em um arquivo contendo representações de tokens, a forma usual de operação de um analisador léxico é por demanda, ou seja, ele tenta compor o token a partir da próxima seqüência de caracteres por solicitação de um outro módulo do compilador, o analisador sintático.

Considere como a seguinte expressão em C é processada pelo compilador:

```
a = a + 2 * b ;
```

O analisador sintático inicia a análise da sentença solicitando ao analisador léxico o próximo token; o retorno indica que o token é um identificador. Como a sentença ainda não está completa, o analisador sintático solicita o próximo token, que é identificado como o operador de atribuição. Os tokens seguintes são solicitados e identificados até o terminador de sentença ser alcançado. Nesse ponto, a informação que o analisador tem é algo da seguinte forma:

`id` `=` `id` `+` `const` `*` `id` `;`

Uma das tarefas do analisador sintático é reconhecer a estrutura da expressão, ou seja:

1. No nível mais alto, este é um comando de atribuição, algo da forma

`lesq` `=` `ldir` `;`

na qual o lado esquerdo (`lesq`) é um identificador (`id`).

2. O lado direito (`ldir`) do comando de atribuição é uma operação de soma, que tem a forma

   `opere` `+` `operd`

   na qual o operando à esquerda (`opere`) é um identificador (`id`).

3. O operando à direita (`operd`) é uma operação de multiplicação, com a forma

   `opere` `*` `operd`

   na qual o operando à esquerda dessa operação é uma constante (`const`) e o operando à direita dessa operação é um identificador (`id`).

Para reconhecer que há, nesse único comando da linguagem de alto nível, três operações elementares que devem ser realizadas e saber em que ordem elas devem ser realizadas, o analisador sintático realiza um processo de reconhecimento da sentença com base na gramática livre de contexto especificada para os comandos da linguagem.

Como nas linguagens naturais, descobrir qual o significado (ou interpretação semântica) associado a uma sentença é uma tarefa bem mais complexa. Algumas tarefas nesse sentido — como verificar se os tipos dos operandos são válidos para a operação — são realizadas pelos compiladores, mas nesse caso as gramáticas não são utilizadas. As abordagens utilizadas para esse tipo de análise são apresentadas no Capítulo 5.

4.1 Reconhecimento de sentenças

O reconhecimento de sentenças, ou *parsing*, é o procedimento que verifica se uma dada sentença pertence à linguagem gerada por uma gramática. Esse procedimento é essencial para um compilador, que deve reconhecer e validar expressões de diversos tipos — declarações, expressões aritméticas, construções de controle de execução — no processo de construção de um código executável equivalente à expressão reconhecida.

Considere uma gramática G_2 que define um subconjunto de expressões aritméticas, apresentada na Figura 4.2. É uma gramática que permite reconhecer expressões envolvendo somas e multiplicações de valores, que pode ser usada para estender o acumulador de valores apresentado no capítulo anterior.

Figura 4.2 Gramática G_2 para expressões com soma e multiplicação

E	$\rightarrow$	$E + E$	1
E	$\rightarrow$	$E \times E$	2
E	$\rightarrow$	(E)	3
E	$\rightarrow$	v	4

As quatro produções dessa gramática contemplam as duas formas básicas de expressões: a soma de duas outras expressões (produção 1) e o produto de duas outras expressões (produção 2). Expressões entre parênteses também são aceitas, desde que o que está entre parênteses seja uma expressão válida (produção 3). A menor expressão válida nessa gramática é, pela produção 4, um token que seja do tipo v (de valor).

Essa gramática tem apenas um único símbolo não-terminal, E, que é portanto também o seu símbolo sentencial. Os símbolos terminais, aqueles que compõem a sentença que deve ser analisada, são os operadores, $+$ e $\times$, os parênteses, (e), e qualquer token do tipo v. Por meio de uma gramática ou expressão regular seria possível determinar como deve ser um token desse tipo para a linguagem. Por exemplo, se valores válidos fossem apenas as letras x ou y ou z, a expressão regular para v seria $x|y|z$. Se fossem valores inteiros, a expressão regular no padrão `lex` [0-9]+ representaria essa classe de tokens.

Com a especificação dessas gramáticas, é possível apresentar como ocorre o reconhecimento de uma sentença. A sentença é válida numa dada gramática apenas quando existe pelo menos uma seqüência de aplicação de produções que permita obter a sentença a partir do símbolo sentencial.

Há duas formas possíveis de derivar essa seqüência de produções para validar uma sentença. No procedimento de reconhecimento descendente, também conhecido como *top-down*, o ponto de partida é o símbolo sentencial. O analisador, com a informação sobre a sentença a ser reconhecida, seleciona uma produção apropriada para aplicar uma derivação e assim obtém uma forma sentencial que se aproxime da sentença. Enquanto há símbolos não-terminais na forma sentencial, produções são selecionadas para levar da forma sentencial até a sentença. Se é possível obter a sentença com essas derivações, então ela é

reconhecida como válida na gramática. Caso contrário, a sentença não faz parte da gramática.

A outra estratégia de validação é o procedimento de reconhecimento ascendente, ou *bottom-up*. Nesse caso, o analisador procura produções cujo lado direito combine com os símbolos que estão na sentença. A substituição que é feita, então, é obter uma forma sentencial a partir da sentença pela substituição do símbolo ou seqüência de símbolos que combinam com o lado direito pelo símbolo não-terminal do lado esquerdo da produção. A forma sentencial assim obtida é analisada e manipulada da mesma forma. A sentença será reconhecida como válida se for possível, ao final dessa seqüência de substituições, obter como resultado apenas o símbolo sentencial.

Como exemplo, considere como um analisador sintático para a gramática G_2, com $v = x|y|z$, pode reconhecer que a expressão $(x + y) \times z$ é válida. O analisador léxico transforma essa expressão na seqüência de tokens

$$\boxed{(}\ \boxed{v}\ \boxed{+}\ \boxed{v}\ \boxed{)}\ \boxed{\times}\ \boxed{v}$$

Para o procedimento descendente, a primeira derivação a ser aplicada é a substituição do símbolo sentencial com a aplicação da produção 2:

$$E \stackrel{2}{\Longrightarrow} E \times E$$

A seleção dessa produção considera que a sentença é, no nível mais alto, uma expressão de multiplicação. Com essa derivação, a forma sentencial obtida aproxima-se da sentença.

Na seqüência, uma nova produção deve ser selecionada para a próxima derivação. Como o operando à esquerda do operador de multiplicação está entre parênteses, a produção 3 é aplicada para substituir o símbolo não-terminal em destaque:

$$\mathbf{E} \times E \stackrel{3}{\Longrightarrow} (E) \times E$$

A expressão que está entre parênteses é, na sentença, uma operação de soma. Portanto, a produção 1 é selecionada para a próxima derivação:

$$(\mathbf{E}) \times E \stackrel{1}{\Longrightarrow} (E + E) \times E$$

Finalmente, para transformar essa forma sentencial na sentença, basta aplicar repetidamente a produção 4:

$$(\mathbf{E} + E) \times E \stackrel{4}{\Longrightarrow} (v + \mathbf{E}) \times E \stackrel{4}{\Longrightarrow} (v + v) \times \mathbf{E} \stackrel{4}{\Longrightarrow} (v + v) \times v$$

Como é possível obter a seqüência de símbolos que compõem a sentença com a aplicação de produções da gramática a partir do símbolo sentencial, então a sentença é reconhecida como válida. Neste exemplo, a sentença foi validada pela seqüência de reconhecimento descendente 2, 3, 1, 4, 4, 4, que é a seqüência das produções que foram utilizadas para derivá-la desde o símbolo sentencial. Caso uma seqüência de reconhecimento não possa ser obtida, então a sentença é inválida e deve ser rejeitada pelo analisador sintático dessa gramática.

A seqüência de reconhecimento para uma sentença não é necessariamente única. Com outra seleção de qual símbolo não-sentencial é escolhido para a próxima derivação, outras seqüências podem ser obtidas. Para esse mesmo exemplo, a seqüência de reconhecimento descendente 2, 3, 4, 1, 4, 4 também leva o símbolo sentencial à sentença sob análise:

$$\begin{aligned} E &\xRightarrow{2} \mathbf{E} \times E \xRightarrow{3} (E) \times \mathbf{E} \xRightarrow{4} (\mathbf{E}) \times v \\ &\xRightarrow{1} (\mathbf{E} + E) \times v \xRightarrow{4} (v + \mathbf{E}) \times v \xRightarrow{4} (v + v) \times v \end{aligned}$$

A mesma sentença poderia ainda ser obtida pela seqüência de reconhecimento descendente 2, 4, 3, 1, 4, 4:

$$\begin{aligned} E &\xRightarrow{2} E \times \mathbf{E} \xRightarrow{4} \mathbf{E} \times v \xRightarrow{3} (\mathbf{E}) \times v \\ &\xRightarrow{1} (E + \mathbf{E}) \times v \xRightarrow{4} (\mathbf{E} + v) \times v \xRightarrow{4} (v + v) \times v \end{aligned}$$

Do mesmo modo, um procedimento de reconhecimento ascendente pode ser aplicado para validar a sentença com uma seqüência de derivações. Nesse caso, as derivações são aplicadas de forma inversa, ou seja, com a substituição, na forma sentencial, de uma seqüência de símbolos que combine com o lado direito de uma produção pelo correspondente lado esquerdo. No exemplo a seguir, a seqüência de símbolos que será substituída em cada derivação é sublinhada:

$$\begin{aligned} (\underline{v} + v) \times v &\xLeftarrow{4} (E + \underline{v}) \times v \xLeftarrow{4} (E + E) \times \underline{v} \xLeftarrow{4} (\underline{E + E}) \times E \\ &\xLeftarrow{1} \underline{(E)} \times E \xLeftarrow{3} \underline{E \times E} \xLeftarrow{2} E \end{aligned}$$

A seqüência para esse reconhecimento ascendente é, portanto, 4, 4, 4, 1, 3, 2. Do mesmo modo que para o reconhecimento descendente, outras seqüências de reconhecimento ascendentes poderiam ser obtidas para a mesma sentença.

4.2 Derivações canônicas

O problema de reconhecimento de uma sentença em uma gramática é essencialmente um problema — dada uma sentença composta por símbolos terminais da linguagem — de encontrar uma seqüência de derivações para ela a partir do símbolo sentencial (a seqüência de reconhecimento) ou, então, indicá-la como inválida na linguagem. O procedimento informal, mostrado na seção anterior, permite um grande número possível de combinações de regras que o torna inadequado para fazer o reconhecimento em qualquer gramática um pouco mais complexa. Felizmente, há procedimentos que restrigem as possibilidades de combinações sem reduzir o poder de expressão das gramáticas.

Quando a forma sentencial tem mais de um símbolo não-terminal, há diversas opções para qual deve ser o próximo símbolo não-terminal a ser substituído na próxima derivação. Uma forma sistemática de selecionar qual símbolo será substituído é estabelecida pelas derivações canônicas. As derivações canônicas permitem obter as sentenças analisando um único símbolo da linguagem a cada passo, com a definição de qual símbolo não-terminal deve ser analisado no momento. As duas possibilidades consideradas são o símbolo que está mais à esquerda ou o símbolo que está mais à direita.

Na derivação canônica mais à esquerda (*leftmost derivation*), a opção é aplicar uma produção da gramática ao símbolo não-terminal que está localizado na posição mais à esquerda da forma sentencial sob análise. Se é possível obter uma seqüência de reconhecimento para a sentença, então é possível obter uma seqüência de reconhecimento que use apenas derivações canônicas mais à esquerda, denominada seqüência de reconhecimento mais à esquerda (*leftmost parse*).

Já na derivação canônica mais à direita (*rightmost derivation*), o símbolo não-terminal mais à direita na forma sentencial é sempre selecionado para ser substituído usando alguma produção da gramática. A seqüência de reconhecimento mais à direita (*rightmost parse*) é dada pelo reverso da seqüência de regras associada à derivação desde o símbolo sentencial — ou seja, como ocorre na construção ascendente, é a seqüência de aplicação de regras para partir da sentença até alcançar o símbolo sentencial.

No exemplo da sentença $(x + y) \times z$ em G_2, a seqüência de reconhecimento mais à esquerda corresponde à primeira que foi obtida, ou seja, 2, 3, 1, 4, 4, 4. Na aplicação da primeira produção da seqüência, a forma sentencial tem apenas o símbolo sentencial; nesse caso, o símbolo não-terminal escolhido é o único. Já ao aplicar a produção 3 havia, na forma sentencial, dois símbolos não-terminais

e o mais à esquerda foi selecionado. Do mesmo modo, ao aplicar a produção 1, o símbolo não-terminal mais à esquerda foi selecionado, assim como para as três aplicações da produção 4.

Nesse mesmo exemplo, a segunda seqüência de derivações que foi apresentada não é uma seqüência canônica, pois os símbolos não-terminais substituídos não são escolhidos de forma sistemática. Já na terceira seqüência, a substituição é sempre do símbolo não-terminal mais à direita. Como foram aplicadas, na ordem, as produções 2, 4, 3, 1, 4 e 4, a seqüência de reconhecimento mais à direita nesse caso é 4, 4, 1, 3, 4, 2.

Observe que uma dada produção é sempre utilizada o mesmo número de vezes nas duas derivações canônicas — a produção 4 é usada três vezes; as outras, uma vez cada.

Ao adotar uma das formas canônicas para o reconhecimento de sentenças, o analisador tem uma única estratégia possível para validar uma sentença. Desse modo, o procedimento de reconhecimento não depende de escolhas aleatórias, o que tornaria sua implementação complexa e potencialmente ineficiente.

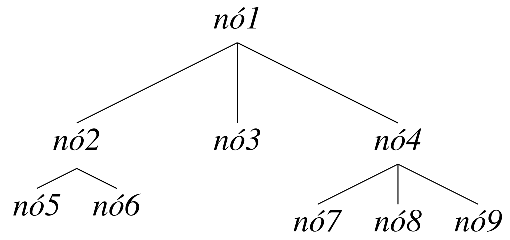
4.3 Árvores sintáticas

O processo de reconhecimento de uma sentença é apenas parte das tarefas do compilador. O resultado desse reconhecimento precisa ser recuperado para que, num momento posterior, o compilador possa produzir o código equivalente na linguagem-alvo. Para tanto, é preciso preservar a informação sobre a seqüência de reconhecimento em uma estrutura de dados apropriada — nesse caso, uma estrutura do tipo árvore.

Uma árvore é uma estrutura de dados que contém um conjunto finito de elementos, usualmente denominados nós, sendo que um desses nós é especialmente designado como nó raiz. O nó raiz é o ponto de entrada para obter todos os elementos da estrutura. Subordinado a um nó podem estar associados subconjuntos disjuntos de nós; cada um desses conjuntos é organizado na forma de uma árvore, denominada subárvore.

A representação gráfica usualmente utilizada para árvores posiciona a raiz no topo do diagrama, com as subárvores abaixo dos nós. A Figura 4.3 apresenta um exemplo no qual *nó1* é a raiz da árvore e esse nó tem três subárvores. A primeira subárvore tem três nós, sendo o *nó2* a raiz dessa subárvore. A segunda subárvore tem apenas um nó, *nó3*, que é a raiz da subárvore e não tem nenhuma subárvore. A terceira subárvore tem quatro nós, com *nó4* como raiz.

Figura 4.3 Representação gráfica de uma árvore



O número de subárvores de um nó determina o grau do nó. No exemplo, o nó *nó1* tem grau 3; o *nó2*, 2; e o *nó3* tem grau 0. O grau da árvore é o maior valor de grau de nó entre todos os nós da árvore; no exemplo, a árvore tem grau 3. Um nó que não tem subárvores, ou seja, cujo grau é 0, é normalmente denominado nó folha da árvore. A árvore da Figura 4.3 tem seis folhas: *nó3, nó5, nó6, nó7, nó8, nó9*. Os nós que são as raízes das subárvores de um nó x são usualmente chamados de nós filhos do nó x; por sua vez, o nó x é o nó pai daqueles nós. No exemplo, *nó5* e *nó6* são filhos do *nó2*; o *nó4* é pai dos nós *nó7, nó8, nó9*. A estrutura de uma árvore é hierárquica, ou seja, cada nó tem apenas um nó pai. A única exceção é o nó raiz da árvore, que não tem um nó pai.

A árvore sintática é uma representação das derivações utilizadas no reconhecimento de uma sentença em uma estrutura de árvore. Na árvore sintática, as folhas da árvore são os símbolos terminais que compõem a sentença. Os nós intermediários da árvore correspondem a símbolos não-terminais, de forma que uma subárvore cuja raiz é um nó P com filhos $s_1, s_2, \ldots, s_n$ pode ocorrer apenas se houver na gramática uma produção $P \rightarrow s_1\ s_2 \ldots\ s_n$. A raiz da árvore sintática para uma sentença válida deve ser um nó com o símbolo sentencial da gramática.

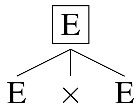
Considere novamente a gramática G_2 (Figura 4.2). Uma árvore sintática para uma sentença dessa gramática só poderá conter subárvores na forma especificada na Figura 4.4.

Figura 4.4 Subárvores para uma árvore sintática em G_2

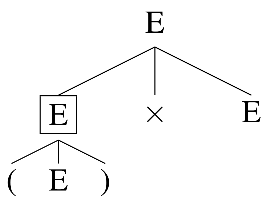


(a) $E \rightarrow E + E$ (b) $E \rightarrow E \times E$ (c) $E \rightarrow (E)$ (d) $E \rightarrow v$

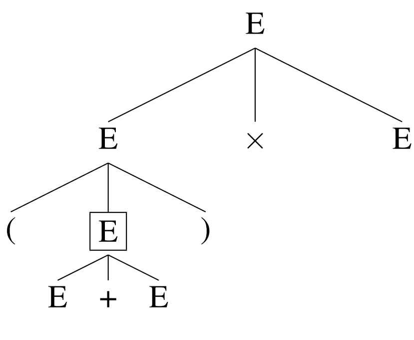
Considere a construção da árvore sintática para a sentença $(v+v)\times v$ em G_2. Pela seqüência de reconhecimento mais à esquerda, as produções são aplicadas na ordem 2, 3, 1, 4, 4, 4. A raiz da árvore é o símbolo sentencial E. Como é o único símbolo não-terminal na árvore, ele é escolhido para a expansão pela aplicação da primeira produção da seqüência, a produção 2. Assim, é criada a subárvore correspondente para o nó com o símbolo não-terminal selecionado para a expansão (em destaque):



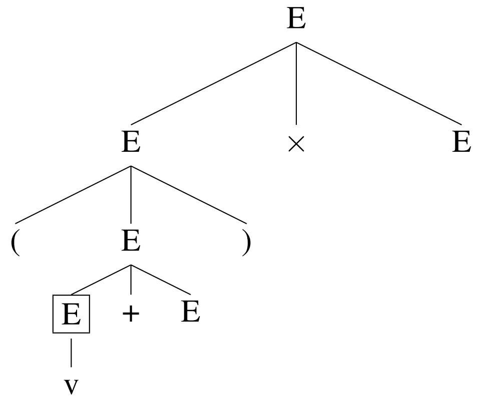
A segunda produção da seqüência é aplicada ao nó folha com símbolo não-terminal mais à esquerda. O resultado é



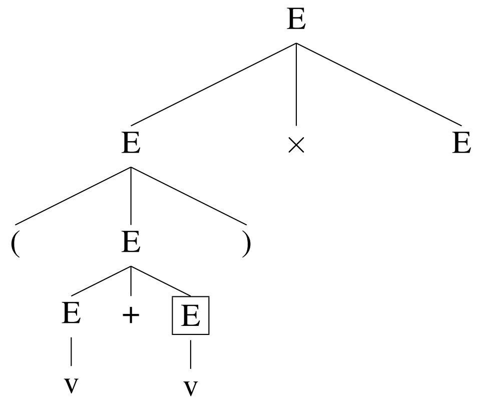
Novamente, o nó folha com símbolo não-terminal mais à esquerda é selecionado para a expansão pela próxima produção da seqüência:



A produção 4 é aplicada ao nó folha com símbolo não-terminal mais à esquerda:



Novamente, a produção 4 é aplicada ao próximo nó folha com símbolo não-terminal mais à esquerda:



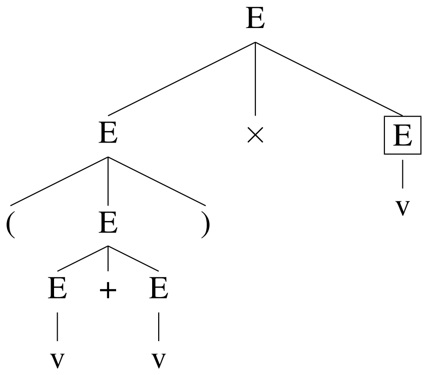
Finalmente, a última aplicação da produção 4 ao último nó folha que ainda tem um símbolo não-terminal resulta na árvore sintática completa para a linguagem, apresentada na Figura 4.5.

Observe que, mesmo que outra seqüência de reconhecimento descendente fosse utilizada, a árvore sintática resultante seria a mesma — apenas a ordem utilizada para a expansão das subárvores teria sido diferente.

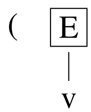
Essa estratégia de construção de árvore, que parte do nó raiz e aplica as produções até que todos os nós folhas tenham os símbolos terminais da sentença, é conhecida como técnica de construção descendente. É também possível construir a árvore das folhas para a raiz, na técnica de construção ascendente. Nesse caso, uma seqüência de reconhecimento ascendente, como a seqüência de reconhecimento mais à direita, é utilizada.

Considere novamente a sentença $(v + v) \times v$, que tem a seqüência de reconhecimento mais à direita 4, 4, 1, 3, 4, 2. O primeiro símbolo da sentença é (,

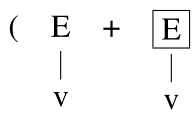
Figura 4.5 Árvore sintática para a sentença $(v + v) \times v$



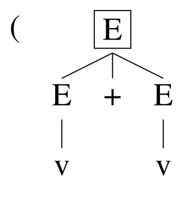
o qual isoladamente não permite a aplicação de nenhuma produção. Então esse símbolo fica pendente, ou seja, sua localização na árvore sintática ainda não está definida. A análise passa ao próximo símbolo, v. Para este, é possível aplicar a primeira produção da seqüência, a produção 4, que permite criar um nó pai para ele com o símbolo não-terminal em destaque:



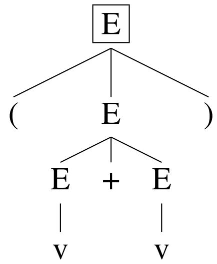
A seqüência de símbolos pendentes agora é (E. Como não há produção aplicável, o próximo símbolo da sentença é analisado, o que resulta na seqüência (E $+$. Ainda é necessário obter o próximo símbolo, que nesse caso é v. Novamente, a produção 4 é aplicada a esse símbolo:



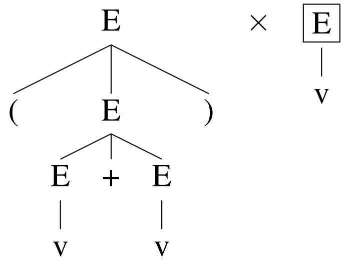
A seqüência dos símbolos pendentes torna-se (E $+$ E. As duas subárvores são reunidas com a aplicação da produção 1 à seqüência de símbolos E $+$ E:



Mais uma vez, a seqüência de símbolos pendentes na árvore é (E. O próximo símbolo,), é incorporado à seqüência. Para os símbolos (E), a produção 3 é aplicada:



Como há símbolos na sentença, a análise prossegue. Com o próximo símbolo da sentença, a seqüência de símbolos pendentes torna-se $E\times$. Como não há seqüência aplicável, o símbolo seguinte, v, é analisado. Com a aplicação da produção 4, que é a próxima na seqüência, um nó pai é criado para esse símbolo:



A seqüência de símbolos pendentes é então $E \ \times \ E$. Com a aplicação da última produção da seqüência, a produção 2, e como não há mais símbolos na sentença, a árvore sintática resultante é a mesma da Figura 4.5.

A árvore sintática obtida pelas duas estratégias de reconhecimento é a mesma, mas é interessante analisar como a estrutura da árvore está relacionada às seqüências de reconhecimento. Para tanto, é preciso entender como os nós de uma árvore podem ser percorridos em uma seqüência.

A informação numa árvore é organizada segundo alguma estratégia de varredura. Em aplicações de busca, por exemplo, árvores são utilizadas para manter a informação ordenada ou para a construção de índices. Nesse caso, os elementos da árvore são mantidos de acordo com a estratégia de varredura intra-ordem. Nessa estratégia, adequada para estruturas de árvore com grau 2 (denominadas árvores binárias), todos os elementos da subárvore esquerda têm valores menores (ou no máximo igual) ao valor do nó raiz; todos os elementos da subárvore

direita têm valores maiores que o valor do nó raiz. Como a estrutura da árvore é recursiva, a estratégia determina precisamente o posicionamento para inserir ou localizar cada valor na árvore.

Outras estratégias de varredura não estão restritas a árvores binárias. Na estratégia pré-ordem, o nó raiz é considerado o primeiro nó da árvore. Na seqüência, os próximos elementos são suas subárvores, da esquerda para a direita. Em cada subárvore, os elementos também são percorridos em pré-ordem.

Considere, por exemplo, como os nós da árvore da Figura 4.3 são organizados na estratégia de varredura pré-ordem. O primeiro nó da seqüência é o nó raiz, *nó1*. A seguir, estão os elementos da primeira subárvore à esquerda. Nesta, o *nó2* é o primeiro elemento, por ser o nó raiz da subárvore. Depois vem a subárvore esquerda (*nó5*) seguida pelos elementos da subárvore direita (*nó6*). Encerrada a varredura dessa subárvore, os próximos elementos vêm da subárvore do meio — nesse caso, há apenas um elemento, *nó3*. Por último, estão os elementos da subárvore mais à direita. Nessa subárvore, o primeiro elemento é o *nó4*, seguido pelos elementos da subárvore da esquerda para a direita: *nó7*, *nó8* e *nó9*. Portanto, a seqüência completa para a varredura dos elementos da árvore nessa estratégia é:

nó1 nó2 nó5 nó6 nó3 nó4 nó7 nó8 nó9

A outra estratégia de varredura é a estratégia pós-ordem, na qual os primeiros elementos são os nós das subárvores, da esquerda para a direita, e o nó raiz é o último elemento da seqüência. Aplicada à árvore da Figura 4.3, a seqüência resultante com a aplicação da estratégia de varredura pós-ordem é

nó5 nó6 nó2 nó3 nó7 nó8 nó9 nó4 nó1

A aplicação de uma dessas duas estratégias de varredura à árvore sintática resulta em uma das seqüências de reconhecimento associada às derivações canônicas. Para a seqüência de reconhecimento mais à esquerda, a estratégia pré-ordem é adotada, ou seja, a primeira produção da seqüência é aquela associada à expansão do nó raiz, seguida pela expansão dos nós das subárvores da esquerda para a direita. Já para a seqüência de reconhecimento mais à direita, a estratégia de varredura utilizada é a pós-ordem, na qual a subárvore mais à esquerda é analisada primeiro, seguida pelas subárvores à direita e com produção associada à expansão do nó raiz por último.

Considere o exemplo da Figura 4.5. A varredura pré-ordem coloca em primeiro lugar a produção 2, usada para a expansão do nó raiz, com o símbolo

sentencial E. A seguir, do lado esquerdo há uma subárvore cuja raiz é um símbolo não-terminal, novamente E. A expansão dessa raiz dá-se pela produção 3. Ainda há símbolo não-terminal abaixo dessa expansão; este é expandido pela produção 1. No próximo nível, há ainda duas subárvores; a da esquerda é expandida pela produção 4, assim como a da direita. Finalizada a análise da subárvore esquerda, é feita a análise da subárvore direita, também expandida pela produção 4. Assim, a varredura pré-ordem resulta na seqüência 2, 3, 1, 4, 4, 4, que é a seqüência de reconhecimento mais à esquerda. De forma similar, a varredura pós-ordem resulta na seqüência 4, 4, 1, 3, 4, 2, que é a seqüência de reconhecimento mais à direita.

4.4 Gramáticas ambíguas

Para uma sentença válida em uma dada gramática, é sempre possível obter suas derivações canônicas ou, de forma equivalente, sua árvore sintática. Idealmente, essas representações devem ser únicas, pois correspondem à interpretação que é dada à sentença.

Considere a árvore sintática da expressão $(v + v) \times v$, apresentada na Figura 4.3. A estrutura da árvore indica que a expressão entre parênteses é avaliada antes da multiplicação. Desse modo, quando o código na linguagem-alvo for gerado, estará claro que a operação de soma precederá a operação de multiplicação, nessa expressão.

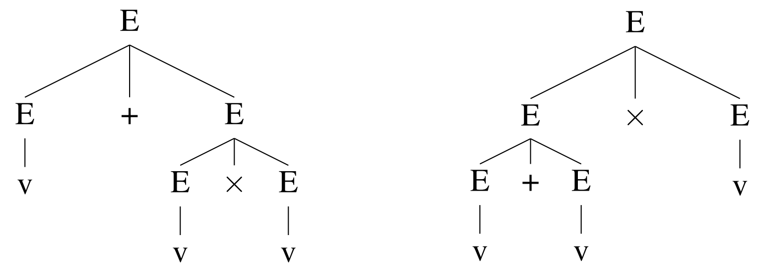
Entretanto, nem sempre a gramática tem suas produções organizadas de modo a oferecer uma única interpretação. Uma gramática ambígua é aquela que permite que uma mesma sentença tenha mais de uma árvore sintática. Em tais situações não há como definir, apenas pelas produções da gramática, qual deve ser a interpretação apropriada para a sentença.

A gramática G_2 é ambígua. Considere, por exemplo, a sua sentença $v + v \times v$, que permite a construção de duas árvores sintáticas distintas. Em uma construção descendente, com derivação canônica mais à esquerda, as produções 1, 4, 2, 4 e 4 levam do símbolo sentencial à sentença:

$$E \Rightarrow E + E \Rightarrow v + E \Rightarrow v + E \times E \Rightarrow v + v \times E \Rightarrow v + v \times v$$

Essa construção está associada à árvore sintática da Figura 4.6(a). Como a subárvore com a expansão da multiplicação está abaixo da operação de soma, essa árvore está associada à interpretação usual da expressão, com a operação de multiplicação com maior precedência.

Figura 4.6 Árvores sintáticas para $v + v \times v$



(a) Equivalente a $v + (v \times v)$ (b) Equivalente a $(v + v) \times v$

No entanto, pelas produções da gramática, nada impede que outra interpretação seja dada à sentença com a atribuição de maior precedência para a operação de adição. Essa outra interpretação corresponde a uma derivação canônica mais à esquerda alternativa:

$$E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow v + E \times E \Rightarrow v + v \times E \Rightarrow v + v \times v$$

Essa seqüência de reconhecimento, 2, 1, 4, 4, 4, está associada à árvore sintática da Figura 4.6(b).

Claramente, gramáticas ambíguas não favorecem o reconhecimento automático de sentenças. O problema é que, dada uma gramática qualquer, não há um procedimento que possa reconhecer se ela é ou não ambígua. Mas, uma vez que se observe que a gramática é ambígua, é possível reescrever suas produções de forma a eliminar a ambigüidade.

Observe qual é a origem da ambigüidade na gramática G_2. O símbolo sentencial é E e todas as produções são igualmente aplicáveis em um dado momento, pois todas têm E no lado esquerdo. Quando a sentença combina as duas operações binárias, soma e multiplicação, as duas produções podem ser igualmente aplicadas.

Como pode ser observado nas árvores sintáticas da Figura 4.6, a precedência menor está associada à operação que está mais próxima da raiz da árvore, ou seja, mais próxima do símbolo sentencial. Se as produções da gramática forem reescritas de forma que a produção da soma esteja mais próxima do símbolo sentencial que a produção da multiplicação, a ambigüidade será eliminada com a atribuição usual das precedências para essas operações.

As produções com maior precedência são aquelas associadas à expansão do token v e às expressões entre parênteses. Seja P um símbolo não-terminal

introduzido para essas duas produções:

$$P \rightarrow (E)$$
$$P \rightarrow v$$

A próxima precedência está associada à operação de multiplicação. Se M é o símbolo não-terminal utilizado para essa operação, então duas produções podem ser utilizadas para representá-la. Uma produção, recursiva, captura a propriedade da associatividade, ou seja, como deve ser tratada uma expressão na qual ocorre mais de uma operação de multiplicação em seqüência:

$$M \rightarrow M \times P$$

A outra produção para a multiplicação marca o fim da recursividade, ou seja, na última expansão o operando da esquerda deve ser um termo simples ou uma expressão entre parênteses:

$$M \rightarrow P$$

Finalmente, para a operação de soma, um par de produções similares é introduzido. A primeira produção, recursiva, é uma expansão para o símbolo sentencial para uma expressão na qual o operador de soma apareça fora de parênteses:

$$E \rightarrow E + M$$

A produção não-recursiva para o símbolo sentencial é:

$$E \rightarrow M$$

Portanto, a gramática G_3, equivalente à gramática G_2 mas sem ambigüidade, é:

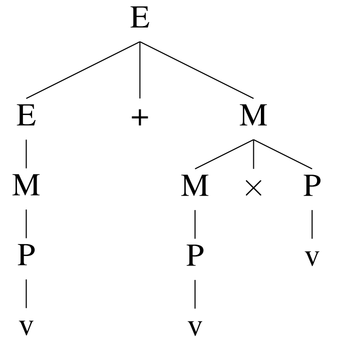
$$\begin{array}{lcll} E & \rightarrow & E + M & 1 \\ E & \rightarrow & M & 2 \\ M & \rightarrow & M \times P & 3 \\ M & \rightarrow & P & 4 \\ P & \rightarrow & (E) & 5 \\ P & \rightarrow & v & 6 \end{array}$$

Desse modo, sempre que houver uma expressão com soma e multiplicação no mesmo nível, a operação de multiplicação estará numa subárvore abaixo da operação de soma, de forma que sua precedência de avaliação seja maior.

Para essa gramática, a única expansão possível para a sentença $v + v \times v$ está associada à seguinte derivação canônica mais à esquerda, cuja árvore sintática está na Figura 4.7:

$$E \Rightarrow E + M \Rightarrow M + M \Rightarrow P + M \Rightarrow v + M$$
$$\Rightarrow v + M \times P \Rightarrow v + P \times P \Rightarrow v + v \times P \Rightarrow v + v \times v$$

Figura 4.7 Árvore sintática para $v + v \times v$ na gramática G_3



Situações de ambigüidade são usuais, na prática de linguagens de programação. Considere a produção em uma gramática de uma linguagem de programação que representa um comando condicional, denotado pelo símbolo não-terminal cond, no qual a cláusula `else` é opcional. A primeira produção está associada à definição do comando `if` sem a parte `else`:

cond → `if` (expr) cmd

A segunda produção define o comando `if` com uma cláusula `else` associada a ele:

cond → `if` (expr) cmd `else` cmd

Como o comando `cmd` pode ser um comando condicional, uma terceira produção é necessária:

cmd → cond

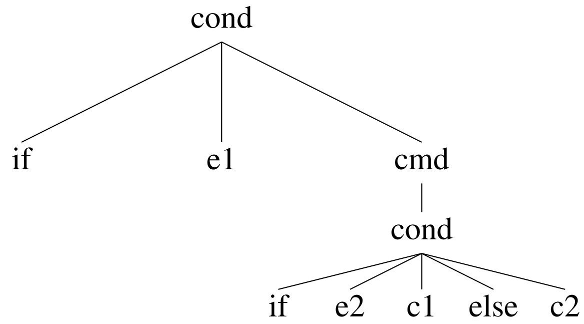
ou seja, o comando presente no interior de um `if`, tanto na parte `then` como na parte `else`, pode ser um outro comando condicional.

Para essas produções, o trecho de código

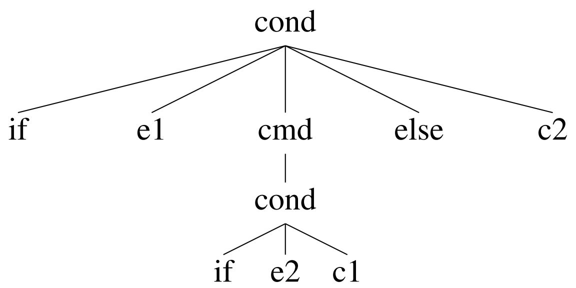
```
if (e1) if (e2) c1 else c2
```

pode dar origem a duas árvores sintáticas distintas. A primeira opção de interpretação para essa sentença está associada ao padrão usualmente adotado em linguagens de programação, que associa o "`else` duvidoso" ao último `if` possível, como representado na árvore sintática da Figura 4.8(a).

Figura 4.8 Árvores sintáticas para um comando `if` ambíguo



(a) Interpretação usual



(b) Outra interpretação possível

A outra árvore sintática, apresentada na Figura 4.8(b), está associada a uma interpretação não-usual em linguagens de programação, porém gramaticalmente correta, que assume que o segundo `if` é que tem a cláusula `else` vazia, que fica associada ao primeiro `if`. Como há duas árvores sintáticas para a mesma sentença, essa gramática para a definição de comando condicional é ambígua.

Na construção de analisadores sintáticos, nem sempre é necessário ter produções não-ambíguas. Em geral, é possível definir alternativas para o tratamento de ambigüidades sem que seja necessário ter a gramática construída sem produções ambíguas, como será visto na Seção 4.6, por meio da atribuição de uma ordem preferencial para a aplicação de produções ou por meio da definição de precedências distintas para operadores.

4.5 Analisadores sintáticos

Dada uma seqüência de símbolos terminais de uma gramática, é possível determinar se essa seqüência é ou não válida nessa gramática por meio da construção de uma árvore sintática. Alternativamente, a validade da sentença poderia ser expressa por meio de uma seqüência de reconhecimento associada a uma derivação canônica, mas nesta seção o foco será na construção da árvore sintática.

Um analisador sintático é um programa construído para uma gramática G que recebe como entrada uma seqüência de símbolos terminais de seu alfabeto. Se a seqüência de símbolos for uma sentença válida, ele constrói sua árvore sintática; caso a sentença não pertença à linguagem descrita por G, ele deve apresentar uma indicação de erro.

Há duas estratégias possíveis para a construção da árvore sintática. Na construção ascendente, o analisador sintático varre a sentença e procura aplicar as produções que permitam substituir seqüências de símbolos da sentença pelo lado esquerdo das produções. A sentença é reconhecida quando, na raiz da árvore sintática, o único símbolo restante é o símbolo sentencial.

Na construção descendente, o objetivo é iniciar a análise com uma lista que contém inicialmente apenas o símbolo sentencial. A partir da análise dos símbolos presentes na sentença, o analisador sintático busca aplicar regras que permitam expandir os símbolos na lista até alcançar a sentença desejada. Nesse caso, o objetivo é obter uma derivação mais à esquerda para uma sentença. Em termos de árvores sintáticas, a construção descendente busca a construção de uma árvore a partir da raiz com o símbolo sentencial e usa a estratégia de varredura pré-ordem para definir o próximo símbolo não-terminal que deve ser considerado para análise e expansão.

Os primeiros compiladores usavam essencialmente dois tipos de analisadores sintáticos. Analisadores baseados em precedência de operadores utilizam a técnica de construção ascendente combinada com informação sobre a precedência e associatividade de operadores da linguagem para guiar suas ações, sendo adequados à análise de expressões aritméticas. Analisadores do tipo descendente recursivo implementam a técnica de construção descendente por meio de um conjunto de rotinas mutuamente recursivas para realizar a análise, sendo normalmente utilizados para outros comandos que não expressões aritméticas.

Na seqüência, serão apresentados dois exemplos de analisadores sintáticos, um para ilustrar a estratégia de construção descendente e outro para ilustrar a estratégia de construção ascendente. Antes, o mecanismo que é comum a ambas as estratégias, o autômato de pilha, é apresentado.

4.5.1 Autômato de pilha

Nos procedimentos informais de reconhecimento de sentenças utilizados nas seções anteriores, a escolha por uma produção da gramática a ser aplicada num dado momento foi direcionada pelo conhecimento sobre a sentença completa e a gramática. Isso foi possível porque, além de a gramática usada nos exemplos ser simples, os seres humanos podem ter essa visão ampla. A máquina, ao contrário, deve operar com a informação limitada que tem em cada momento.

Da mesma forma que foi possível, para a análise léxica, definir uma máquina de estados que permitiu construir o autômato finito para reconhecer automaticamente strings de uma linguagem regular, é desejável ter um autômato que possa ser utilizado para automatizar o procedimento de análise sintática. Entretanto, pelas características das linguagens livres de contexto, não é possível utilizar o mesmo tipo de autômato que foi usado para reconhecer tokens. O principal motivo é a propriedade da auto-incorporação, que demanda no processamento que alguma forma de memória esteja disponível. O autômato finito opera sem nenhum tipo de memória, pois considera apenas o estado corrente e o próximo símbolo da string.

Um autômato de pilha é a estrutura formal que incorpora a memória necessária para o reconhecimento de sentenças em linguagens livres de contexto. Formalmente, um autômato de pilha é uma sêxtupla $M = (K, \Sigma, \Gamma, \delta, s, F)$, na qual os elementos são:

1. o conjunto finito de estados, K;
2. o alfabeto de entrada finito, Σ;
3. o alfabeto de pilha finito, Γ;
4. a função de transição, $\delta : K \times \Sigma \times \Gamma \rightarrow K \times \Gamma$;
5. o estado inicial, s, $s \in K$;
6. o conjunto de estados finais, F, com $F \subseteq K$.

Em relação à definição do autômato finito, as principais diferenças são a introdução da pilha e o fato de que a transição leva em conta, além do estado corrente e do símbolo da string, o símbolo que está no topo da pilha. Além disso, o efeito da transição também pode alterar a pilha, com a remoção ou com a introdução de novos símbolos na pilha.

Uma estrutura de dados do tipo pilha é uma estrutura linear com restrição na política de acesso aos seus elementos. Na pilha, o único elemento que pode ser manipulado num dado momento é o último elemento que foi inserido na coleção. Uma pilha implementa a política de acesso LIFO (*last in, first out*), ou seja, o último elemento que entra é o primeiro que sai. Alguns aspectos relacionados à implementação de uma estrutura desse tipo em C++ são analisados a seguir.

A biblioteca STL de C++ oferece algumas estruturas lineares, como a definida pela classe `vector`, apresentada na Seção 3.3.2. Outra estrutura linear de STL é o `deque`, que opera eficientemente com inserções e remoções em ambas as extremidades da estrutura. Além dessas classes parametrizadas, a biblioteca STL também oferece alguns adaptadores de coleções, que usam alguma coleção internamente para oferecer um elemento com maior grau de abstração. Esse é o caso da classe parametrizada `stack`.

Um `stack` de STL implementa uma pilha para qualquer tipo de elemento que seja especificado na declaração da estrutura. Por exemplo, para criar uma pilha de valores do tipo `int`, a seguinte declaração é utilizada:

```
#include <stack>
...
stack<int> simbolos;
```

A classe `stack` oferece as operações `push`, para inserir um elemento no topo da pilha, e `pop`, para remover o elemento no topo da pilha. Além dessas operações, que são as operações essenciais para a manipulação da estrutura de pilha, a classe tem também os métodos `top`, para inspecionar o elemento que está no topo; `empty`, para testar se a pilha está vazia e `size`, para obter a quantidade de elementos na pilha. O exemplo a seguir ilustra a utilização desses métodos.

```
for (int pos=0; pos<3; ++pos)
    simbolos.push(pos);

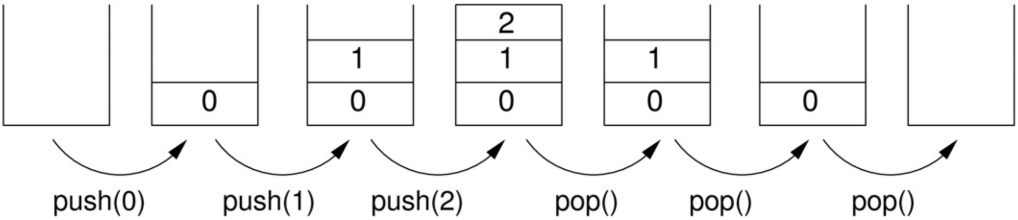
while (! simbolos.empty()) {
    cout << "Pilha tem " << simbolos.size()
         << " elementos, topo: " << simbolos.top()
         << endl;
    simbolos.pop();
}
```

Esse exemplo, quando executado, apresenta na tela o resultado

```
Pilha tem 3 elementos, topo: 2
Pilha tem 2 elementos, topo: 1
Pilha tem 1 elementos, topo: 0
```

A Figura 4.9 ilustra como o estado da pilha `simbolos` evolui nesse exemplo à medida que as operações que alteram o seu estado, `push` e `pop`, são executadas.

Figura 4.9 Evolução do estado da pilha



Embora possa usar qualquer estrutura linear como elemento interno, a implementação padrão de `stack` utiliza o `deque`. Caso se deseje utilizar outra estrutura, como um vetor, ela pode ser especificada como um segundo parâmetro no momento da declaração:

```
stack<int, vector<int> > outraPilha;
```

Como poderá ser observado nas próximas seções, as funções providas pela classe `stack` de STL atendem perfeitamente as necessidades da estrutura de pilha usada como auxiliar no reconhecimento de sentenças de uma linguagem livre de contexto.

Da mesma forma que para os autômatos finitos, para os quais uma tabela de transições é a estrutura de dados que suporta a operação dos analisadores léxicos, para os autômatos de pilha é necessário ter uma tabela para a realização da análise sintática. A estratégia de construção dessas tabelas a partir da gramática depende do tipo de analisador que é desenvolvido. Nas próximas seções, dois exemplos ilustrarão a construção dessas tabelas e seu uso no procedimento de reconhecimento de sentenças em linguagens livres de contexto.

4.5.2 Analisador sintático preditivo

O analisador sintático preditivo é baseado na técnica de construção descendente. Assim, no processo de reconhecimento de uma sentença, ele inicia com o símbolo sentencial e, pela análise dos próximos símbolos da sentença, decide qual produção deve ser aplicada para expandir o símbolo não-terminal corrente. Se a cada aplicação de uma expansão a informação é atualizada na estrutura da árvore sintática, ao final do processo de reconhecimento de uma sentença válida o resultado será a árvore sintática completa para a sentença.

Pela sua característica de operação, o analisador sintático preditivo não pode operar com gramáticas que tenham produções com recursão à esquerda. Se for este o caso, é necessário inicialmente reescrever essas produções de modo a substituir a recursão à esquerda por recursão à direita.

Com uma gramática com suas produções na forma apropriada, é possível construir a tabela sintática do analisador. Essa tabela é a estrutura de informação utilizada pelo analisador para conduzir a operação de reconhecimento de sentenças pelo autômato de pilha.

Conversão de produções recursivas

Caso haja na gramática alguma produção recursiva na qual o símbolo não-terminal do lado esquerdo da produção seja também o primeiro símbolo do lado direito, ou seja, uma produção com recursão à esquerda, então o primeiro passo na construção do analisador sintático preditivo é reescrever essas produções de modo a eliminar essa forma de recursão. A estratégia é usar a recursão à direita, ou seja, onde o símbolo do lado esquerdo da produção aparece como o último do lado direito.

Seja A o símbolo não-terminal de uma produção recursiva à esquerda, ou seja, de uma produção da forma

$$A \rightarrow A\beta$$

na qual β é uma seqüência qualquer de símbolos não iniciada pelo símbolo A. Além dessa produção, deve haver outra produção não-recursiva para o símbolo A. Se δ é uma outra seqüência de símbolos não iniciada por A, então essa produção tem a forma

$$A \rightarrow \delta$$

As formas sentenciais associadas à expansão do símbolo não-terminal A,

como descrito por essas duas produções, são

$$\begin{aligned}
&A \Rightarrow \delta\\
&A \Rightarrow A\beta \Rightarrow \delta\beta\\
&A \Rightarrow A\beta \Rightarrow A\beta\beta \Rightarrow \delta\beta\beta\\
&A \Rightarrow A\beta \Rightarrow A\beta\beta \Rightarrow A\beta\beta\beta \Rightarrow \delta\beta\beta\beta\\
&A \Rightarrow \ldots
\end{aligned}$$

ou seja, são seqüências iniciadas pela seqüência δ seguidas por zero ou mais ocorrências da seqüência de símbolos β.

Para expressar as mesmas formas sentenciais por meio de produções sem recursão à esquerda, o par de produções mostrado é substituído por três produções da seguinte maneira. Primeiro, uma produção indica que a expansão de A começa com δ seguida por outros símbolos, associados a um novo símbolo não-terminal A':

$$A \rightarrow \delta A'$$

A seguir, é preciso descrever que A' é uma repetição de zero ou mais ocorrências de β. Repetição nas formas sentenciais de uma linguagem é expressa nas produções de uma gramática por meio de recursão. Como recursão à esquerda não é desejada, a recursão à direita é utilizada:

$$A' \rightarrow \beta A'$$

Todo símbolo não-terminal que tem uma produção recursiva precisa também de uma produção que encerre a recursão. Nesse caso, como é possível ter nenhuma ocorrência de β na expansão de A, uma produção com expansão para a string vazia, ε, é utilizada:

$$A' \rightarrow \varepsilon$$

Essas três produções descrevem exatamente as mesmas formas sentenciais que o par de produções originais cuja recursão era à esquerda:

$$\begin{aligned}
&A \Rightarrow \delta A' \Rightarrow \delta\\
&A \Rightarrow \delta A' \Rightarrow \delta\beta A' \Rightarrow \delta\beta\\
&A \Rightarrow \delta A' \Rightarrow \delta\beta A' \Rightarrow \delta\beta\beta A' \Rightarrow \delta\beta\beta\\
&A \Rightarrow \delta A' \Rightarrow \delta\beta A' \Rightarrow \delta\beta\beta A' \Rightarrow \delta\beta\beta\beta A' \Rightarrow \delta\beta\beta\beta\\
&A \Rightarrow \ldots
\end{aligned}$$

Considere como exemplo a gramática G_3, para expressões de soma e multiplicação sem ambigüidades. A primeira produção de G_3 apresenta a recursão à esquerda:

$$E \rightarrow E + M$$

Para fazer a conversão, é preciso considerar também qual é a produção não-recursiva para o símbolo E, que é a produção 2:

$$E \rightarrow M$$

Nesse caso, a seqüência de símbolos que corresponde a β é $+M$ e a que corresponde a δ é M. Portanto, o par de produções originais pode ser substituído por

$$\begin{aligned} E &\rightarrow ME' \\ E' &\rightarrow +ME' \\ E' &\rightarrow \varepsilon \end{aligned}$$

Da mesma forma, o par de produções seguintes, para o símbolo não-terminal M, também apresenta a recursão à esquerda na primeira produção:

$$\begin{aligned} M &\rightarrow M \times P \\ M &\rightarrow P \end{aligned}$$

Nesse caso, $\beta = \times P$ e $\delta = P$. Portanto, as produções reescritas são

$$\begin{aligned} M &\rightarrow PM' \\ M' &\rightarrow \times PM' \\ M' &\rightarrow \varepsilon \end{aligned}$$

As duas últimas produções não apresentam recursão e portanto não precisam ser modificadas. Assim, a gramática G_4, que é equivalente às gramáticas G_2 e G_3 mas não tem ambigüidade e com produções recursivas que têm apenas

recursão à direita, tem produções

$$
\begin{aligned}
E &\rightarrow ME' \\
E' &\rightarrow +ME' \\
E' &\rightarrow \varepsilon \\
M &\rightarrow PM' \\
M' &\rightarrow \times PM' \\
M' &\rightarrow \varepsilon \\
P &\rightarrow (E) \\
P &\rightarrow v
\end{aligned}
$$

Seus símbolos terminais são os mesmos, $V_t = \{+, \times, (,), v\}$, mas o conjunto de símbolos não-terminais, que em G_2 continha apenas E e em G_3 era $\{E, M, P\}$, agora contém cinco elementos, $V_n = \{E, E', M, M', P\}$. O símbolo sentencial ainda é E.

Construção da tabela sintática

A tabela sintática é a estrutura que descreve a operação do autômato de pilha para o reconhecimento de sentenças. Como o analisador sintático preditivo é baseado na técnica de construção descendente, o objetivo é partir do símbolo sentencial e expandi-lo até alcançar a sentença sob análise. A expansão do símbolo sentencial é a informação mantida na pilha, que é esvaziada à medida que os símbolos da sentença são reconhecidos.

Desse modo, a tabela sintática é organizada de forma a indicar, para um dado símbolo não-terminal que está no topo da pilha e para um dado símbolo terminal que é o próximo da sentença, qual produção deve ser aplicada. Assim, a tabela sintática tem uma linha para cada símbolo não-terminal e uma coluna para cada símbolo terminal. O conteúdo P_i no cruzamento da linha L com a coluna c indica que a produção P_i é aplicada para expandir o símbolo L que está no topo da pilha quando o próximo símbolo da sentença é c. Além das colunas para os símbolos terminais, há uma coluna para um símbolo adicional, que é o delimitador de sentença, aqui indicado pelo símbolo $\boxed{\$}$.

Para construir a tabela sintática para uma gramática sem produções recursivas à esquerda, deve-se analisar cada uma das suas produções. O objetivo dessa análise das produções é determinar, no momento da expansão do símbolo não-terminal que está no topo da pilha — ou, de modo equivalente, a criação de uma

subárvore para esse símbolo na árvore sintática —, qual é a produção que deve ser escolhida.

Seja uma produção $A \rightarrow \alpha$ da gramática, na qual A é um símbolo não-terminal da gramática e α é qualquer seqüência de símbolos, terminais ou não. Para que essa produção seja aplicada para expandir o símbolo A que está no topo da pilha, é preciso que o próximo símbolo da sentença possa iniciar a expansão de A. Assim, é preciso obter o conjunto de símbolos terminais T que podem iniciar uma seqüência a partir de α.

Se a seqüência α for iniciada por um símbolo terminal t, então o conjunto T é composto apenas por esse próprio símbolo. Nesse caso, a produção $A \rightarrow \alpha$ é registrada na linha A, coluna t da tabela sintática.

Caso contrário, o primeiro símbolo da seqüência é um símbolo não-terminal. Assim, é necessário avaliar quais são as possíveis expansões desse símbolo e descobrir quais são os símbolos terminais que podem iniciar a expansão de A por essa produção. Nesse caso, pode ser que o conjunto T tenha mais de um símbolo terminal. Para cada símbolo $t \in T$, a tabela sintática recebe a entrada com o valor $A \rightarrow \alpha$ no cruzamento da linha A com a coluna t.

Eventualmente, pode ser que a expansão do primeiro símbolo não-terminal de α leve à string vazia. Como a informação na tabela sintática é determinística e não pode conter referências à string vazia, caso isso ocorra, é preciso analisar quais são as possíveis expansões do símbolo seguinte em α, da mesma forma que descrito anteriormente para o primeiro símbolo do lado direito da produção.

Uma situação diferente ocorre quando a produção é da forma $A \rightarrow \varepsilon$. Nesse caso, o analisador precisa conhecer quais são os símbolos terminais que podem ocorrer após A, pois são estes que estarão na sentença caso a produção seja aplicada. Por isso, é necessário analisar quais são os símbolos terminais que podem aparecer imediatamente após o símbolo não-terminal A.

Uma primeira regra para descobrir esses elementos trata a situação particular do símbolo sentencial, que tem como símbolo terminal que pode ocorrer imediatamente após ele o delimitador de sentença, $\boxed{\$}$. Assim, $\boxed{\$}$ faz parte do conjunto de elementos que pode aparecer após o símbolo sentencial.

A outra condição que permite descobrir quais são esses símbolos é quando o símbolo A aparece no lado direito de uma produção e há outros símbolos após A, como na produção

$$X \rightarrow \alpha A \beta$$

na qual α e β são duas seqüências quaisquer de símbolos. Se a seqüência β tiver como um primeiro símbolo um símbolo terminal, este então faz parte do conjunto de elementos que pode aparecer após o símbolo não-terminal A. Caso

a seqüência β inicie com um símbolo não-terminal B, então todos os símbolos terminais que podem iniciar a expansão de B devem fazer parte do conjunto de elementos que pode aparecer após o símbolo A.

A última condição que deve ser analisada para a definição desse conjunto ocorre quando o símbolo A aparece ao final da produção, como em

$$X \rightarrow \alpha A$$

ou quando, no caso anterior, a expansão de todos os símbolos na seqüência β pode resultar na string vazia. Nessa condição, se existe um símbolo terminal t que pode aparecer numa forma sentencial após o símbolo não-terminal X, então esse símbolo pode também aparecer após A. Assim, todos os símbolos terminais que podem aparecer após X também devem ser incluídos no conjunto de símbolos terminais que podem aparecer após A.

Se S é o conjunto de todos os símbolos terminais que podem aparecer após o símbolo não-terminal A, então a produção $A \rightarrow \varepsilon$ será incluída na tabela sintática na linha A, coluna s para cada $s \in S$.

Considere a construção da tabela sintática do analisador sintático preditivo para a gramática G_4, a gramática para expressões aritméticas de soma e subtração sem ambigüidades e sem produções recursivas à esquerda. Para essa construção, cada uma das produções deve ser analisada.

A primeira produção da gramática é

$$E \rightarrow ME' \qquad \text{(P1)}$$

Como o lado direito da produção começa com o símbolo não-terminal M, os símbolos terminais que podem iniciar a expansão de E serão os mesmos que podem iniciar a expansão de M. Pela análise da produção que tem M do lado esquerdo, seu lado direito começa com o símbolo não-terminal P. Portanto, os símbolos terminais que podem iniciar a expansão de M e, conseqüentemente, também de E, serão os mesmos que podem iniciar a expansão de P. Há duas produções que têm P do lado esquerdo. A primeira delas tem como primeiro símbolo do lado direito o símbolo terminal $($ e a outra produção, o símbolo terminal v. Portanto, são esses os dois símbolos terminais que podem iniciar a expansão de E.

Como resultado dessa análise, a produção P1 aparece na tabela sintática duas vezes, na linha E, coluna $($, e na linha E, coluna v.

Da mesma forma, é preciso analisar a segunda produção,

$$E' \rightarrow +ME' \qquad \text{(P2)}$$

Nesse caso, como o lado direito já tem como primeiro símbolo um símbolo terminal, este é o único que pode iniciar a expansão de E' por essa produção. Portanto, a produção P2 só aparece na tabela na linha E', coluna $+$.

A terceira produção é

$$E' \rightarrow \varepsilon \qquad \text{(P3)}$$

Como o lado direito dessa produção é a string vazia, então é preciso analisar quais são os símbolos terminais que podem aparecer à direita de E' para determinar quando essa produção poderá ser aplicada. Nas produções da gramática, não há nenhum lado direito no qual haja algum símbolo à direita de E', mas há uma produção — P1 — na qual E' é o último símbolo do lado direito e o lado esquerdo não é E', mas E. Então, é preciso descobrir quais são os símbolos terminais que podem aparecer após E, pois eles também serão os símbolos terminais que podem aparecer após E'.

Como E é o símbolo sentencial da gramática, o primeiro desses símbolos é o delimitador de sentenças, $\boxed{\$}$. O outro símbolo terminal desse conjunto é $)$, pois há uma produção com lado direito no qual esse símbolo ocorre após E. Assim, a produção P3 aparece na tabela sintática na linha E', coluna $\boxed{\$}$ e na linha E', coluna $)$.

A quarta produção é

$$M \rightarrow PM' \qquad \text{(P4)}$$

Para essa produção, os símbolos que podem iniciar a expansão de M são os mesmos que podem iniciar a expansão de P. Como já analisado, esses símbolos são $($ e v e, portanto, a produção P4 aparece na tabela sintática na linha M, coluna $($ e na linha M, coluna v.

A produção seguinte é

$$M' \rightarrow \times PM' \qquad \text{(P5)}$$

Da mesma forma que a produção P2, nesse caso o lado direito já começa com um símbolo terminal. Portanto, a produção P5 aparece na tabela apenas na linha M', coluna $\times$.

A sexta produção, como P3, tem o lado direito igual à string vazia:

$$M' \rightarrow \varepsilon \qquad \text{(P6)}$$

Novamente, é preciso analisar quais são os símbolos que podem aparecer numa forma sentencial imediatamente à direita de M'. Pela produção P4, esses serão os mesmos que podem aparecer à direita de M. Nas produções P1 e P2, E'

é o símbolo que aparece à direita de M. Nesse caso, há duas situações que devem ser analisadas. Na primeira situação, os símbolos terminais que podem dar início à expansão de E' serão os símbolos que podem aparecer à direita de M e, portanto, de M'. Nesse caso, apenas o símbolo $+$ é inserido no conjunto dos símbolos que podem aparecer à direita de M'. A segunda situação precisa ser analisada porque há uma expansão de E' para a string vazia e, portanto, os símbolos que podem aparecer à direita de E' também poderão aparecer à direita de M'. Como já analisado para a produção P3, esses símbolos são) e $\boxed{\$}$. Desse modo, a produção P6 aparece três vezes na tabela sintática: na linha M', coluna $+$, na linha M', coluna), e na linha M', coluna $\boxed{\$}$.

A primeira produção para o símbolo P é

$$P \rightarrow (E) \qquad \text{(P7)}$$

Como o lado direito inicia com um símbolo terminal, essa produção só aparece na tabela na linha P, coluna (. Do mesmo modo, a última produção da gramática

$$P \rightarrow v \qquad \text{(P8)}$$

também só produz uma entrada na tabela sintática, na linha P, coluna v.

Quando todas as produções da gramática são analisadas, a tabela sintática está completa. Nessa tabela há uma linha para cada símbolo não-terminal e uma coluna para cada símbolo terminal e também para o símbolo delimitador de sentença. No caso da gramática G_4, a tabela sintática é apresentada na Tabela 4.1.

Tabela 4.1 Tabela sintática para a gramática G_4

	$+$	$\times$	(	)	v	$\boxed{\$}$
E			P1		P1	
E'	P2			P3		P3
M			P4		P4	
M'	P6	P5		P6		P6
P			P7		P8	

Nem sempre, entretanto, a tabela sintática pode ser construída como nesse exemplo, no qual não há nenhuma situação de conflito. Considere a seguinte

gramática, com símbolos terminais $\{0, 1\}$, símbolos não-terminais $\{A, B\}$, símbolo sentencial A e produções:

$$A \rightarrow 1A1$$
$$A \rightarrow B0$$
$$B \rightarrow 1$$

Essa é uma gramática livre de contexto com produções sem recursão à esquerda. As sentenças associadas a essa gramática são da forma

$$A \Rightarrow B0 \Rightarrow 10$$
$$A \Rightarrow 1A1 \Rightarrow 1B01 \Rightarrow 1101$$
$$A \Rightarrow 1A1 \Rightarrow 11A11 \Rightarrow 11B011 \Rightarrow 111011$$

ou seja, sentenças com n ocorrências do símbolo 1, seguidas por uma ocorrência do símbolo 0 e mais $n - 1$ ocorrências do símbolo 1.

Na construção da tabela sintática, a primeira produção dessa gramática está associada com a linha A, coluna 1. A terceira produção também tem sua posição na tabela, linha B, coluna 1, definida diretamente pela ocorrência do símbolo terminal na primeira posição do lado direito. A segunda produção, entretanto, indica que a expansão de A por essa produção está associada aos símbolos terminais que iniciam a expansão de B que, pela terceira produção, é 1. Assim, na posição da tabela linha A, coluna 1, há duas entradas, uma para a primeira e outra para a segunda produção.

Como o analisador poderia decidir qual produção usar se o símbolo no topo da pilha for A e o próximo símbolo da sentença for 1? Analisadores para gramáticas como essa não têm como tomar essa decisão apenas com a informação de um símbolo da sentença; eles teriam de espiar qual é o símbolo seguinte da sentença para poder tomar a decisão sobre qual das duas produções aplicar.

Uma gramática que permite a construção do analisador de construção descendente que sempre consegue descobrir qual produção aplicar apenas com a informação do topo da pilha e com um símbolo da sentença é denominada gramática LL(1). Esse nome indica que a varredura da sentença ocorre da esquerda para a direita (*left-to-right*) e que no reconhecimento da sentença é utilizada a derivação canônica mais à esquerda (*leftmost derivation*), com um único símbolo da sentença (*lookahead*) analisado para a tomada de decisão no processo de reconhecimento.

Uma gramática com duas produções $A \rightarrow \alpha$ e $A \rightarrow \beta$, nas quais α e β são duas seqüências quaisquer de símbolos, é LL(1) se apresentar as seguintes propriedades:

1. α e β não podem iniciar com o mesmo símbolo terminal ou com símbolos não-terminais cujas expansões comecem com o mesmo símbolo terminal;

2. No máximo, uma das duas seqüências, ou α ou β, pode ter expansões que levem à string vazia;

3. Se uma das duas produções tem o lado direito que leva à string vazia, a outra produção não pode ter no início de sua expansão um símbolo terminal que também possa ocorrer imediatamente à direita de A.

Se a gramática é LL(1), então a tabela sintática para o analisador sintático preditivo pode ser construída sem entradas duplicadas e o analisador usado de forma determinística para o reconhecimento de suas sentenças da forma como descrito a seguir.

Reconhecimento de sentença

O procedimento de reconhecimento de uma sentença com o analisador sintático preditivo é uma aplicação do autômato de pilha. A sentença a ser analisada é uma seqüência de tokens que é fornecida pelo analisador léxico, mas para fins da descrição desse procedimento pode ser abstraída como uma lista de tokens que é entregue completa ao analisador.

O estado inicial para a operação do analisador contém essa lista de tokens que corresponde à sentença sob análise, acrescida ao final do símbolo delimitador de sentença. O delimitador de sentença também é inserido como o primeiro elemento da pilha. Como a construção é descendente, a pilha recebe ainda o símbolo sentencial da gramática.

A operação do autômato analisa, a cada transição, o símbolo que está no topo da pilha e o primeiro símbolo da lista de tokens. As possibilidades de operação nesse analisador são:

1. O símbolo no topo da pilha é $\boxed{\$}$ e o primeiro símbolo da lista de tokens também é $\boxed{\$}$: a sentença foi reconhecida.

2. O símbolo no topo da pilha é um símbolo terminal t e o primeiro símbolo da lista de tokens é o mesmo símbolo terminal t: os símbolos são eliminados do topo da pilha e do início da sentença.

3. O símbolo no topo da pilha é um símbolo não-terminal A, o primeiro símbolo da lista de tokens é um símbolo terminal t e há na tabela sintática

uma produção na linha A, coluna t: o elemento no topo da pilha é retirado e os símbolos do lado direito da produção indicada na tabela são inseridos, em ordem reversa (da direita para a esquerda), na pilha. Não há alteração na lista de tokens.

Qualquer outra situação que não essas representa uma condição de erro na análise, ou seja, a sentença não foi reconhecida.

Considere como o procedimento é utilizado para reconhecer a sentença $v + v \times v$ em G_4. O estado inicial das estruturas do analisador é:

Pilha:

E
$

Lista de tokens:

v	$+v \times v$ $

Se o objetivo do reconhecimento da sentença for a construção da árvore sintática, esse estado inicial corresponde a uma árvore apenas com o elemento raiz:

$$E$$

Nessa condição, o analisador procura na tabela sintática, na linha E, coluna v, qual produção deve ser aplicada. A indicação nesse caso é a produção P1, $E \to ME'$. Portanto, o símbolo no topo da pilha é substituído pelos dois símbolos do lado direito:

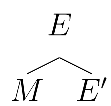
Pilha:

M
E'
$

Lista de tokens:

v	$+v \times v$ $

Na construção da árvore sintática, o efeito é a criação dos nós filhos correspondentes à expansão por essa produção:



Novamente, o analisador busca na tabela qual produção é indicada, dessa vez na linha M, coluna v. A produção encontrada é P4, $M \to PM'$. Portanto, o símbolo no topo da pilha é substituído pelos dois símbolos do lado direito dessa produção e a lista de tokens permanece inalterada:

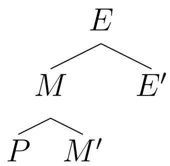
Pilha:

P
M'
E'
$\$$

Lista de tokens:

v	$+v \times v$ $\$$

O efeito na construção da árvore sintática é a expansão do nó M:



Nessa condição, a tabela sintática indica, na linha P, coluna v, a aplicação da produção P8, $P \rightarrow v$. Portanto, somente a pilha é alterada novamente, com a remoção do elemento P no topo e a inserção do lado direito da produção, v:

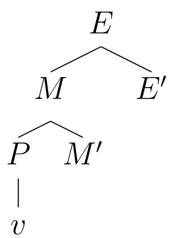
Pilha:

v
M'
E'
$\$$

Lista de tokens:

v	$+v \times v$ $\$$

Na árvore sintática, o nó P é expandido por essa produção:



Nesse ponto, o autômato realiza a primeira transição, ou seja, o analisador reconheceu o primeiro símbolo da sentença. Assim, o topo da pilha e o primeiro token da lista são eliminados e o novo estado do analisador é:

Pilha:

M'
E'
$\$$

Lista de tokens:

$+$	$v \times v$ $\$$

Essa operação não tem efeito na construção da árvore sintática, pois nenhuma produção foi aplicada.

Na linha M', coluna $+$ da tabela sintática, há a indicação para a aplicação da produção P6, $M' \rightarrow \varepsilon$. Nesse caso, ocorre apenas a eliminação do elemento no topo da pilha, pois a substituição pela string vazia significa que nenhum símbolo ocorre no lado direito dessa produção. O novo estado do analisador é:

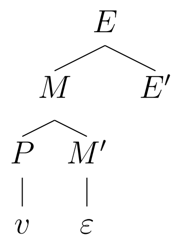
Pilha:

E'
\$

Lista de tokens:

$+$	$v \times v$ \$

A aplicação dessa produção tem o efeito de expandir o nó M' na construção da árvore sintática:



Na linha E', coluna $+$, a tabela sintática indica que a produção P2, $E' \rightarrow +ME'$ deve ser aplicada. Com a substituição do topo da pilha por esses elementos, o novo estado das estruturas do analisador é:

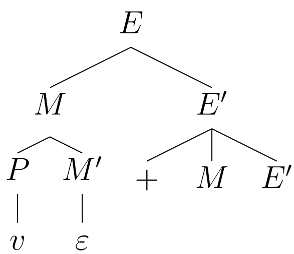
Pilha:

$+$
M
E'
\$

Lista de tokens:

$+$	$v \times v$ \$

A ação correspondente na construção da árvore sintática é a expansão do nó E':



Novamente, a ação do analisador é a remoção do símbolo terminal, nesse caso $+$, do topo da pilha e da lista de tokens. O estado resultante é:

Pilha:

M
E'
$\$$

Lista de tokens:

v	$\times v$	$\$$

A árvore sintática não é alterada por essa operação do analisador.

A produção P4, $M \rightarrow PM'$, está na linha M, coluna v da tabela sintática. Sua aplicação leva a:

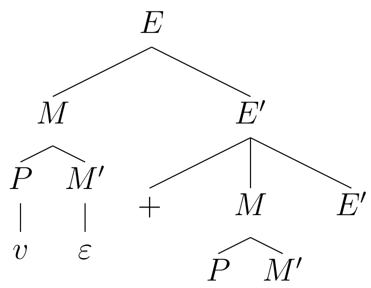
Pilha:

P
M'
E'
$\$$

Lista de tokens:

v	$\times v$	$\$$

Na árvore sintática, há a expansão do nó M:



Novamente, a produção P8, $P \rightarrow v$, é aplicada:

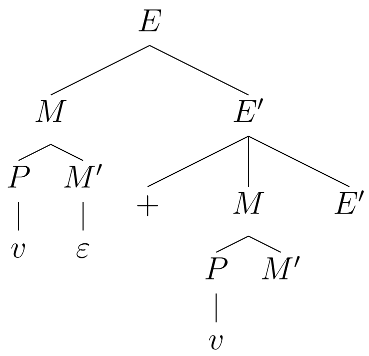
Pilha:

v
M'
E'
$\$$

Lista de tokens:

v	$\times v$	$\$$

Na árvore sintática, o nó P é expandido:



Nessa condição, o analisador elimina o símbolo terminal v do topo da pilha e da lista de tokens, sem alterar a árvore sintática:

Pilha:

M'
E'
$

Lista de tokens:

$\times$	v $

Com M' no topo da pilha e $\times$ como próximo símbolo da sentença, a produção P5, $M' \rightarrow \times P M'$ é aplicada:

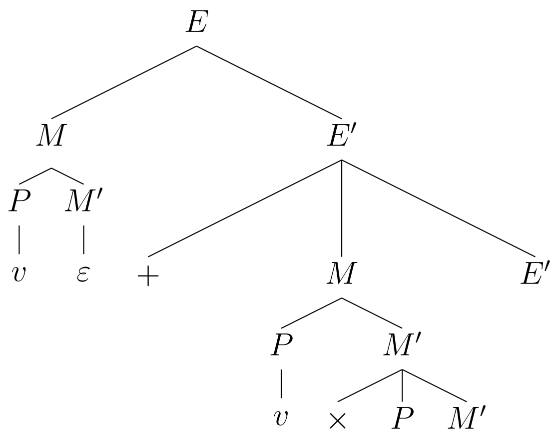
Pilha:

$\times$
P
M'
E'
$

Lista de tokens:

$\times$	v $

Conseqüentemente, o analisador expande o nó M' com essa produção na árvore sintática:



Nessa condição, o símbolo terminal $\times$ é eliminado do topo da pilha e da lista de tokens:

Pilha:

P
M'
E'
\$

Lista de tokens:

v	\$

Novamente, com P no topo da pilha e próximo símbolo v, a produção P8 é aplicada:

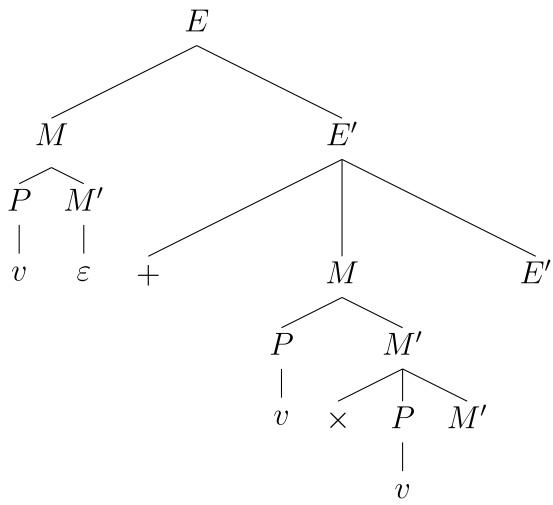
Pilha:

v
M'
E'
\$

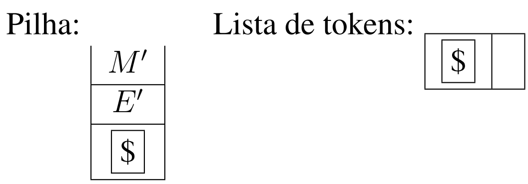
Lista de tokens:

v	\$

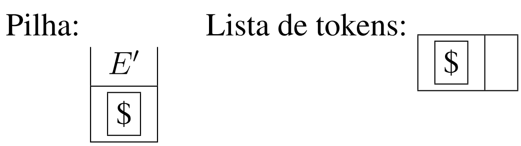
Na árvore sintática, há a expansão do nó P:



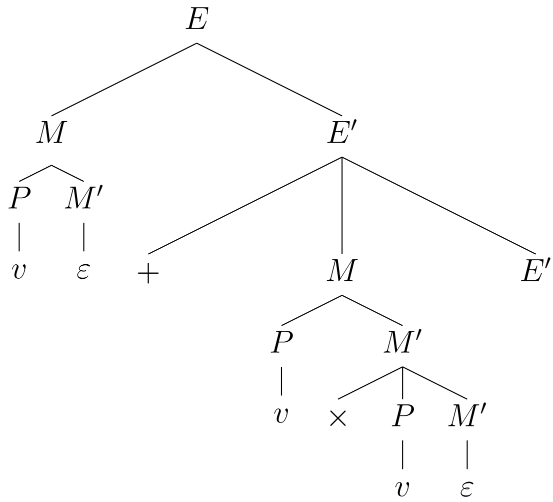
O analisador elimina o último símbolo terminal, v, da sentença e do topo da pilha:



Com M' na pilha e próximo token $\boxed{\$}$, a produção P6, $M' \to \varepsilon$ é aplicada, ou seja, o símbolo M' é eliminado do topo da pilha:

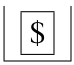


Na construção da árvore sintática, o nó M' é expandido por essa produção:

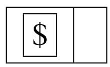


Com E' no topo da pilha e próximo símbolo $\boxed{\$}$, a produção P3, $E' \rightarrow \varepsilon$ é aplicada, ou seja, o símbolo E' é eliminado do topo da pilha:

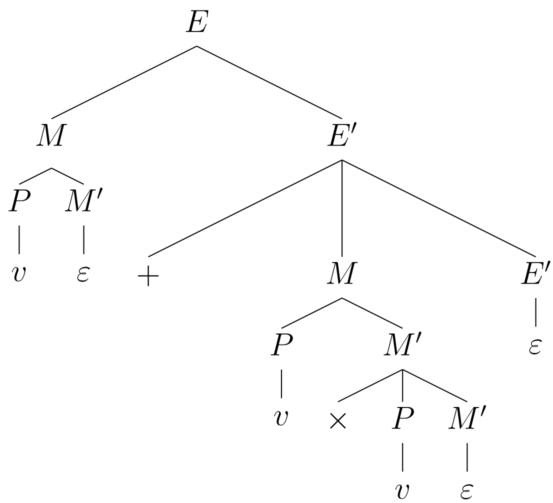
Pilha:



Lista de tokens:



Ao atingir essa condição de suas estruturas, com o símbolo delimitador de sentença $\boxed{\$}$ no topo da pilha e também como símbolo corrente da lista de tokens, o analisador reconhece que chegou ao fim da análise com o reconhecimento da sentença. Na árvore sintática, a aplicação da produção $E' \rightarrow \varepsilon$ expande o último nó não-terminal:



As folhas dessa árvore sintática são, na seqüência da esquerda para a direita, $v\varepsilon{+}v{\times}v\varepsilon\varepsilon$. Sem a representação da string vazia, essa árvore sintática representa exatamente a estrutura completa da sentença $v + v \times v$ na gramática G_4.

No procedimento de construção descendente da árvore sintática, o analisador sintático escolhe para expansão um nó folha com um símbolo não-terminal. No início há apenas o símbolo sentencial no nó raiz, mas depois é comum que haja mais de um símbolo não-terminal nas folhas da árvore. Nesses casos, o nó escolhido para a expansão foi sempre o nó folha com o símbolo não-terminal que estava mais à esquerda. Como esse símbolo corresponde ao primeiro símbolo não-terminal na forma sentencial a cada momento, esse analisador realiza o reconhecimento pela derivação canônica mais à esquerda.

4.5.3 Analisador de precedência fraca

O analisador sintático preditivo faz o reconhecimento de uma sentença pela estratégia de construção descendente da árvore sintática. A outra estratégia possível é realizar o reconhecimento pela técnica de construção ascendente, que será aqui ilustrada pelo analisador sintático de precedência fraca. Esse analisador também tem limitações relativas às características da gramática, mas é um dos mais simples que permite ilustrar a estratégia de reconhecimento por construção ascendente.

Da mesma forma que com o analisador sintático preditivo, a operação do analisador de precedência fraca está associada a um autômato de pilha. A estrutura e as operações desse autômato também são representadas por meio de uma tabela, a tabela de deslocamento e redução, gerada a partir da análise da gramática.

O procedimento de reconhecimento de uma sentença por esse tipo de analisador verifica a seqüência de tokens e decide, de acordo com o autômato codificado na tabela, se o símbolo deve ser deslocado para a pilha ou se uma produção deve ser aplicada para reduzir uma parte da seqüência para um símbolo não-terminal. Por esse motivo, a tabela para o analisador é denominada deslocamento (de um token para a pilha) e redução (dos símbolos no topo da pilha).

Na construção da árvore sintática, a aplicação de uma produção para realizar uma redução equivale à introdução de um nó pai a partir dos nós filhos que foram reconhecidos. À medida que as subárvores são criadas e combinadas, a árvore sintática é construída em direção à raiz. Dessa forma, o procedimento de reconhecimento é encerrado com sucesso quando o nó raiz da árvore, após a varredura de todos os tokens, for o símbolo sentencial.

Gramáticas de precedência fraca

O analisador de precedência fraca é adequado para reconhecer sentenças de uma classe restrita de gramáticas livres de contexto. As propriedades de uma gramática apropriada a esse tipo de analisador são aqui apresentadas.

Verificar se a gramática obedece a algumas propriedades necessárias para a construção do analisador sintático de precedência fraca é simples. Por exemplo, a gramática não pode conter nenhuma produção cujo lado direito seja a string vazia. Outra propriedade simples de verificar apenas pela inspeção das produções é a de que a gramática deve ser unicamente inversível — não pode haver duas produções que tenham o mesmo lado direito.

Há uma outra propriedade que não tem verificação tão direta, mas é ainda facilmente observável: a gramática deve ser livre de ciclos. Se A é um símbolo não-terminal na gramática, não deve existir nenhuma derivação $A \stackrel{+}{\Rightarrow} A$, ou seja, não pode haver uma seqüência de derivação que produza como resultado o mesmo símbolo A.

As demais propriedades dependem da avaliação de relações um pouco mais elaboradas. Para auxiliar na definição dessas relações, dois conjuntos são associados a cada símbolo não-terminal da gramática. Se X é o símbolo não-terminal sob análise, o conjunto $ESQ(X)$ contém todos os símbolos que podem dar início à expansão de X por meio de uma quantidade qualquer de derivações. Formalmente,

$$ESQ(X) = \{Y \in V_T \cup V_N | X \stackrel{+}{\Rightarrow} Y\alpha, \text{ para } \alpha \in (V_T \cup V_N)*\}$$

Similarmente, o conjunto $DIR(X)$ contém os símbolos que podem terminar alguma derivação do símbolo não-terminal X,

$$DIR(X) = \{Y \in V_T \cup V_N | X \stackrel{+}{\Rightarrow} \alpha Y, \text{ para } \alpha \in (V_T \cup V_N)*\}$$

Por exemplo, considere uma gramática G_5 com símbolos não-terminais $V_N = \{S, X\}$, com S como símbolo sentencial, símbolos terminais $V_T = \{a, b, c, d, e\}$ e produções $P = \{S \rightarrow aSb, S \rightarrow Xc, X \rightarrow d, X \rightarrow e\}$. Os conjuntos ESQ e DIR para os dois símbolos não-terminais de G_5 são

$$\begin{aligned} ESQ(S) &= \{X, a, d, e\} \\ DIR(S) &= \{b, c\} \\ ESQ(X) &= \{d, e\} \\ DIR(X) &= \{d, e\} \end{aligned}$$

Com essas definições, é possível definir as relações de precedência de Wirth-Weber, que são analisadas para avaliar se uma gramática é ou não de precedência fraca. Seja $G = (V_N, V_T, P, S)$ uma gramática livre de contexto; as relações $\approx$, $\prec$ e $\succ$ são definidas para dois símbolos $X, Y \in V_N \cup V_T$ pelas seguintes regras:

1. $X \approx Y$ se existe pelo menos uma produção em G cujo lado direito tenha X imediatamente antes de Y,

$$A \rightarrow \alpha XY\beta$$

sendo α e β quaisquer strings em G.

2. $X \prec Y$ se existe um símbolo não-terminal $Z \in V_N$ tal que

$$X \approx Z \quad \text{e} \quad Y \in ESQ(Z)$$

3. $X \succ a$, $a \in V_T$, se uma das duas condições é verdadeira:

   (a) Existe um símbolo não-terminal $Z \in V_N$ tal que

$$Z \approx a \quad \text{e} \quad X \in DIR(Z)$$

   ou

   (b) Existem símbolos não-terminais $Z_1, Z_2 \in V_N$ tal que

$$Z_1 \approx Z_2 \quad \text{e} \quad X \in DIR(Z_1) \quad \text{e} \quad a \in ESQ(Z_2)$$

Agora é possível estabelecer as condições restantes para determinar quando uma gramática é de precedência fraca. Se a relação $X \succ Y$ ocorre para dois símbolos X, Y de uma gramática G, então não pode ocorrer na gramática a relação $X \prec Y$ ou $X \approx Y$. Além disso, quando duas produções de G terminam com uma mesma string β, como em $A \rightarrow \alpha X\beta$ e $B \rightarrow \beta$, sendo que α é uma string em G e X um símbolo qualquer da gramática, então o par de símbolos X, B não pode estar associado por nenhuma das relações de Wirth-Weber. Se essas duas condições são observadas juntamente com aquelas apresentadas anteriormente, então G é uma gramática de precedência fraca.

Considere novamente a gramática G_5, usada para exemplificar as definições dos conjuntos ESQ e DIR. Por inspeção, é fácil verificar que as três primeiras condições são atendidas — não há produções com lados direitos iguais ou vazios e tampouco há produções que produzam ciclos nas derivações. Com a análise

das relações de Wirth-Weber, será possível determinar se a gramática é ou não de precedência fraca.

Inicialmente, é preciso determinar para quais pares de símbolo ocorre a relação $\approx$, o que é feito por inspeção dos lados direitos das produções. Da produção $S \rightarrow aSb$, verifica-se que $a \approx S$ e $S \approx b$. Da outra produção para esse símbolo, $S \rightarrow Xc$, obtém-se a relação $X \approx c$. As duas produções para o símbolo X têm apenas um símbolo do lado direito, portanto não derivam nenhuma ocorrência dessa relação.

Para obter as ocorrências da relação $\prec$, conforme a regra 2, é preciso analisar todas as relações $\approx$ que tenham do lado direito um símbolo não-terminal. Nesse caso, há apenas a relação $a \approx S$. Pela regra, $a \prec Z$ para cada símbolo $Z \in ESQ(S)$. Portanto, obtêm-se as ocorrências $a \prec X$, $a \prec a$, $a \prec d$ e $a \prec e$.

Similarmente, para obter as ocorrências da relação $\succ$ como definido pela regra 3, é preciso avaliar duas situações associadas a relações $\approx$. A primeira ocorre quando há um símbolo não-terminal do lado esquerdo da relação; nesse caso, há $S \approx b$ e $X \approx c$. Da relação $S \approx b$ e do conjunto $DIR(S) = \{b, c\}$, obtêm-se as relações $b \succ b$ e $c \succ b$. Do mesmo modo, da relação $X \approx c$ e do conjunto $DIR(X) = \{d, e\}$ obtêm-se as relações $d \succ c$ e $e \succ c$. A segunda situação que deve ser analisada para obter ocorrências dessa relação é quando há símbolos não-terminais em ambos os lados da relação $\approx$. No caso particular dessa gramática, tal situação não ocorre.

A Tabela 4.2 resume as relações obtidas para a gramática G_5.

Tabela 4.2 Relações de Wirth-Weber para a gramática G_5

	S	X	a	b	c	d	e
S				$\approx$			
X					$\approx$		
a	$\approx$	$\prec$	$\prec$			$\prec$	$\prec$
b				$\succ$			
c				$\succ$			
d					$\succ$		
e					$\succ$		

Como não há nenhuma posição da tabela na qual a relação $\succ$ apareça juntamente com uma das outras duas relações, a primeira condição é atendida. Adicionalmente, como não há nenhum par de produções que terminem com

a mesma string, então a segunda condição nem precisa ser avaliada — G_5 é uma gramática de precedência fraca.

A tabela de deslocamento e redução

A base para a operação de reconhecimento nesse tipo de analisador é a tabela de deslocamento e redução (DR). Essa tabela determina, a partir do símbolo no topo da pilha e do próximo símbolo terminal na sentença, se o próximo passo da análise é ler o próximo símbolo, reduzir os símbolos já lidos ou, em caso de uma sentença inválida, se não há ação a ser tomada.

Para construir a tabela DR, as produções da gramática são analisadas para obter suas relações de precedência de Wirth-Weber. Para simplificar o processo de reconhecimento de sentenças, duas regras adicionais são consideradas para obter relações que envolvem o símbolo delimitador de sentença, $\boxed{\$}$. Se S é o símbolo sentencial da gramática, a primeira regra estabelece que $\boxed{\$} \prec X$ para cada símbolo $X \in ESQ(S)$; a segunda, que $X \succ \boxed{\$}$ para cada símbolo $X \in DIR(S)$.

A tabela DR está organizada para direcionar o processo de reconhecimento pela análise do conteúdo do topo da pilha X, que pode ser um símbolo qualquer da gramática, e do próximo símbolo ainda não analisado da sentença t, um símbolo terminal. Se existir a relação $X \prec t$ ou $X \approx t$ entre X e o símbolo terminal t, então a tabela DR deverá ter uma entrada na tabela na linha X, coluna t, para indicar que a ação com X no topo da pilha e t como próximo token deve ser a inserção de t na pilha (o deslocamento, D). Caso a relação existente seja $X \succ t$, então a ação indicada na linha X, coluna t, é a redução (R), ou seja, os símbolos no topo da pilha combinam com o lado direito de uma produção e devem ser substituídos pelo seu lado esquerdo.

Pela sua organização, as colunas da tabela DR contêm apenas símbolos terminais, acrescidos do delimitador de sentença. Já as linhas contêm todos os símbolos — terminais, não-terminais e o delimitador de sentença. Sua construção torna-se trivial se uma tabela com as relações de precedência de Wirth-Weber é construída primeiro.

Tome como exemplo a gramática G_5 apresentada anteriormente. É preciso inicialmente aplicar as duas regras adicionais para o delimitador de sentença. Nesse caso, obtêm-se as relações $\boxed{\$} \prec X$, $\boxed{\$} \prec a$, $\boxed{\$} \prec d$, $\boxed{\$} \prec e$, $b \succ \boxed{\$}$ e $c \succ \boxed{\$}$. Com as relações já obtidas anteriormente, a tabela resultante é apresentada na Tabela 4.3.

Na tabela, a entrada "D" indica que a ação deve ser de empilhar o próximo

Tabela 4.3 Tabela de deslocamento e redução para a gramática G_5

	a	b	c	d	e	\$
S		D				
X			D			
a	D			D	D	
b		R				R
c		R				R
d			R			
e			R			
\$	D			D	D	

símbolo da sentença (deslocamento), enquanto a entrada "R" determina a redução dos símbolos no topo da pilha que combinam com o lado direito de uma produção. Para as entradas em branco não há uma ação que possa ser tomada que leve ao reconhecimento da sentença; portanto, tal situação indica uma condição de erro no reconhecimento.

Reconhecimento de sentença

O analisador de precedência fraca trabalha com duas estruturas de dados auxiliares, além da tabela de deslocamento e redução. A primeira delas é a lista de símbolos terminais a analisar, que contém inicialmente a sentença submetida à análise delimitada ao final pelo símbolo \$. A outra estrutura é uma pilha com os símbolos já analisados, os quais podem ter sido eventualmente substituídos por símbolos não-terminais pela aplicação de produções da gramática. Portanto, a pilha pode conter qualquer símbolo, terminal ou não-terminal, do alfabeto da gramática.

Para o procedimento de reconhecimento de uma sentença, a pilha é iniciada apenas com o símbolo delimitador de sentença. A lista de símbolos terminais que compõem a sentença também recebe ao final esse símbolo. A cada passo do procedimento, a tabela DR é consultada com o símbolo no topo da pilha como índice para a linha e o símbolo no início da lista como índice para a coluna. Se a entrada nessa célula estiver vazia, então o analisador não tem a indicação de nenhuma ação a realizar — é uma condição de erro, que é usualmente reportada com uma indicação da posição na sentença em que o analisador ficou sem ação.

Se, por outro lado, a entrada nessa célula indicar deslocamento, então o símbolo no início da lista é removido dessa estrutura e inserido no topo da pilha.

Se a entrada na tabela DR para o par de símbolos sob análise indicar redução, então todos os símbolos no topo da pilha que combinam com o lado direito da produção são removidos da pilha e substituídos pelo símbolo do lado esquerdo da produção, que é empilhado. Sob o ponto de vista da construção da árvore sintática, essa ação corresponde a criar uma subárvore cuja raiz é o símbolo que foi inserido na pilha e cujos filhos são os elementos retirados da pilha.

Dessa forma, torna-se evidente se o procedimento é concluído com sucesso: quando o único símbolo restante na pilha for o símbolo sentencial, além dos dois delimitadores de sentença introduzidos pelo procedimento. Essa situação equivale à obtenção da árvore sintática completa, com o símbolo sentencial na raiz e nenhum símbolo da sentença sem pertencer à árvore.

Considere, por exemplo, o reconhecimento da sentença $aadcbb$ na gramática G_5. O estado inicial das estruturas do analisador é

Pilha:

\$

Lista de tokens:

a	$a\ d\ c\ b\ b$ \$

A ação indicada na linha \$, coluna a da tabela DR é de deslocamento. Portanto, o símbolo a é retirado do início da lista de tokens e inserido no topo da pilha:

Pilha:

a
\$

Lista de tokens:

a	$d\ c\ b\ b$ \$

Novamente, para a linha a, coluna a, a ação indicada é de deslocamento. Portanto,

Pilha:

a
a
\$

Lista de tokens:

d	$c\ b\ b$ \$

Do mesmo modo, para a linha a, coluna d, a ação é de deslocamento:

Pilha:

d
a
a
$

Lista de tokens:

c	$b\ b$	$

Já para a linha d, coluna c, a ação indicada é redução. A produção cujo lado direito combina com o topo da pilha é $X \rightarrow d$. Portanto, o símbolo d é retirado da pilha e substituído pelo símbolo X:

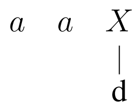
Pilha:

X
a
a
$

Lista de tokens:

c	$b\ b$	$

Nesse ponto do procedimento de reconhecimento, a árvore sintática recebe seu primeiro nó interno, correspondente à aplicação dessa produção para o primeiro subconjunto de símbolos da sentença que foi reconhecido. Dos símbolos da sentença analisados até este ponto, apenas o símbolo d foi incorporado à árvore sintática — há dois símbolos a que ainda devem esperar pelo reconhecimento:



Com o símbolo X no topo da pilha e o símbolo c no início da lista de tokens, a próxima ação é deslocamento:

Pilha:

c
X
a
a
$

Lista de tokens:

b	b	$

A próxima ação indicada na tabela DR, para a linha c, coluna b, é redução. A produção aplicável nesse caso é $S \rightarrow Xc$. Portanto, os dois símbolos no topo da pilha são substituídos pelo lado esquerdo da produção:

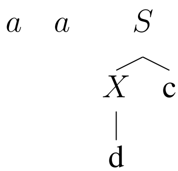
Pilha:

S
a
a
$

Lista de tokens:

b	b	$

Com essa ação, mais um segmento da árvore sintática foi construído:



Embora S seja o símbolo sentencial, ainda há símbolos a analisar e portanto o procedimento de reconhecimento continua. Para a linha S, coluna, b, a ação é deslocamento:

Pilha:

b
S
a
a
$

Lista de tokens:

b	$

Para a linha b, coluna b, a ação é de redução — nesse caso, pela produção $S \rightarrow aSb$:

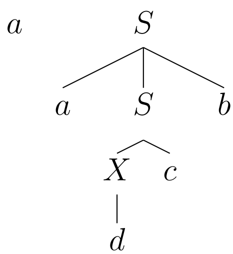
Pilha:

S
a
$

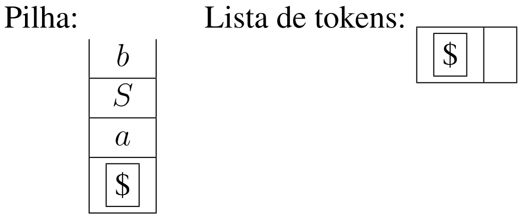
Lista de tokens:

b	$

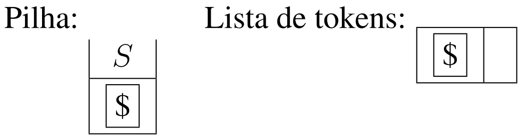
Na construção ascendente da árvore sintática, o efeito dessa redução é:



Na seqüência, o último símbolo da lista de tokens é deslocado para a pilha, conforme indicação na tabela DR para a linha S, coluna b:

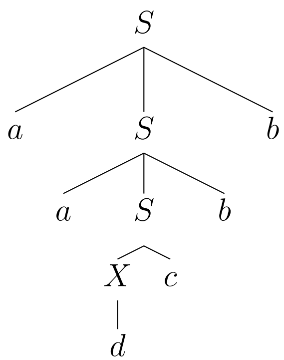


Para a linha b, coluna $\boxed{\$}$, a ação indicada na tabela é redução. A produção aplicável é novamente $S \rightarrow aSb$:



A condição de terminação do procedimento de reconhecimento foi alcançada, nesse caso com sucesso: o único símbolo da gramática que restou na pilha, após o consumo de todos os símbolos terminais na lista de tokens, foi o símbolo sentencial.

A aplicação da última produção também conclui a construção da árvore sintática, com todos os símbolos da sentença incorporados e com o símbolo sentencial na raiz:



Exemplo

Considere como um analisador de precedência fraca pode ser construído para a gramática de expressões G_3. O primeiro passo é avaliar se a gramática atende às condições para a construção do analisador de precedência fraca. Para tanto, é

preciso obter as relações de Wirth-Weber, verificar se não há conflitos entre as relações para nenhum par de símbolos e, para as produções que terminam com o mesmo símbolo, verificar se não há ambigüidade entre o símbolo anterior e uma possível redução do símbolo.

Para obter as relações de Wirth-Weber, é preciso obter os conjuntos ESQ e DIR para cada símbolo não-terminal de G_3:

$$
\begin{array}{ll}
ESQ(E) = \{M, P, (, v\} & DIR(E) = \{M, P,), v\} \\
ESQ(M) = \{P, (, v\} & DIR(M) = \{P, (, v\} \\
ESQ(P) = \{(, v\} & DIR(P) = \{), v\}
\end{array}
$$

A primeira relação, $\approx$, é obtida por inspeção das produções. Para G_3 obtêm-se:

$$
\begin{array}{ll}
E \approx + & + \approx M \\
M \approx \times & \times \approx P \\
(\approx E & E \approx \,) \\
E \approx v &
\end{array}
$$

Para a segunda relação, $\prec$, é preciso analisar as relações $\approx$ que têm símbolos não-terminais do lado direito e os elementos do conjunto ESQ desses símbolos. Como $+ \approx M$ e $ESQ(M) = \{P, (, v\}$, então $+ \prec P$, $+ \prec ($ e $+ \prec v$. Do mesmo modo obtêm-se novas relações a partir de $\times \approx P$ e de $(\approx E$. Portanto, todas as relações $\prec$ são:

$$
\begin{array}{ll}
+ \prec P & + \prec (\\
+ \prec v & \times \prec P \\
\times \prec v & (\prec M \\
(\prec P & (\prec (\\
(\prec v &
\end{array}
$$

A terceira relação, $\succ$, é obtida a partir da análise das relações $\approx$ entre um símbolo não-terminal e um símbolo terminal ou entre dois símbolos não-terminais. Há quatro relações no primeiro caso: $E \approx +$, $M \approx \times$, $E \approx \,)$ e $E \approx v$, das quais são derivadas as relações:

$$
\begin{array}{ll}
M \succ + & P \succ + \\
) \succ + & v \succ + \\
P \succ \times &) \succ \times \\
v \succ \times & M \succ \,) \\
P \succ \,) &) \succ \,) \\
v \succ \,) & M \succ v \\
P \succ v &) \succ v \\
v \succ v &
\end{array}
$$

Como não há nenhuma relação $\approx$ entre dois símbolos não-terminais, este é o conjunto completo de relações $\succ$ para G_3.

A Tabela 4.4 apresenta o conjunto completo das relações de Wirth-Weber entre os símbolos da gramática G_3.

Tabela 4.4 Relações de Wirth-Weber para a gramática G_3

	E	M	P	$+$	$\times$	$($	$)$	v
E				$\approx$			$\approx$	$\approx$
M				$\succ$	$\approx$		$\succ$	$\succ$
P				$\succ$	$\succ$		$\succ$	$\succ$
$+$		$\approx$	$\prec$			$\prec$		$\prec$
$\times$			$\approx\prec$					$\prec$
$($	$\approx$	$\prec$	$\prec$			$\prec$		$\prec$
$)$				$\succ$	$\succ$		$\succ$	$\succ$
v				$\succ$	$\succ$		$\succ$	$\succ$

Como não há nenhum par de símbolos que esteja relacionado simultaneamente pela relação $\succ$ e por alguma outra relação, a primeira condição para que a gramática seja de precedência fraca é satisfeita.

A segunda condição demanda a análise das produções que terminam com o mesmo símbolo. O primeiro par de produções que precisa ser analisado é $E \to E + M$ e $E \to M$, pois ambas terminam com o símbolo M. A condição que precisa ser analisada é se há alguma relação entre o símbolo que precede M na primeira produção, nesse caso $+$, e o lado esquerdo da segunda produção, nesse caso E. Como pode ser observado pela entrada vazia na linha $+$ com coluna E, não há nenhuma relação entre $+$ e E. Então, pela análise desse par de produções, a gramática é de precedência fraca.

A mesma análise deve ser realizada para o par de produções $M \rightarrow M \times P$ e $M \rightarrow P$. Da mesma forma, é possível observar que não há relação entre o símbolo que precede P na primeira produção, $\times$, e o símbolo não-terminal do lado esquerdo da segunda produção, M, pois a entrada na linha $\times$ e coluna M está vazia. Como todas as regras que terminam com o mesmo símbolo foram analisadas e não resultaram em conflitos, a gramática G_3 é de precedência fraca.

Para a construção da tabela DR, é preciso incluir as relações envolvendo o símbolo terminador de sentença $\boxed{\$}$ e os símbolos que pertencem aos conjuntos ESQ e DIR do símbolo sentencial E. Como $ESQ(E) = \{M, P, (, v\}$, então

$$\boxed{\$} \prec M$$

$$\boxed{\$} \prec P$$

$$\boxed{\$} \prec (\,$$

$$\boxed{\$} \prec v$$

Do mesmo modo, como $DIR(E) = \{M, P,), v\}$, então

$$M \succ \boxed{\$}$$

$$P \succ \boxed{\$}$$

$$) \succ \boxed{\$}$$

$$v \succ \boxed{\$}$$

Assim, a Tabela 4.5 apresenta a tabela DR para G_3.

Considere como a tabela é utilizada no procedimento de reconhecimento da sentença $v + v \times v$. No estado inicial, a pilha contém apenas o símbolo delimitador de sentença e a lista contém todos os tokens da sentença e o delimitador de sentença ao final:

Pilha: $\boxed{\$}$ Lista de tokens: $\boxed{v \;|\; + v \times v \boxed{\$}}$

A primeira ação indica deslocamento, portanto o próximo estado é

Pilha: v / $\boxed{\$}$ Lista de tokens: $\boxed{+ \;|\; v \times v \boxed{\$}}$

Tabela 4.5 Tabela DR para a gramática G_3

	$+$	$\times$	$($	$)$	v	$\boxed{\$}$
E	D			D	D	
M	R	D		R	R	R
P	R	R		R	R	R
$+$			D		D	
$\times$					D	
$($			D		D	
$)$	R	R		R	R	R
v	R	R		R	R	R
$\boxed{\$}$			D		D	

Essa ação tem um efeito na construção da árvore sintática, que é a criação de um nó folha para a árvore com o token v.

Para esse estado, com v no topo da pilha e $+$ como próximo token, a ação indicada é redução, nesse caso pela produção 6, $P \rightarrow v$, a única aplicável. Assim, o símbolo v no topo da pilha (o lado direito da produção) é substituído pelo símbolo não-terminal P, o lado esquerdo da produção:

Pilha: P / $\boxed{\$}$ Lista de tokens: $+ \mid v \times v \boxed{\$}$

Na construção da árvore sintática, essa ação corresponde à criação do nó pai para o nó que havia sido criado pela ação anterior:

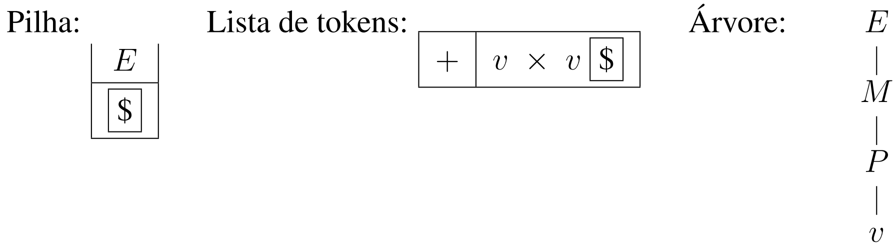
P
|
v

Com o símbolo P no topo da pilha e $+$ como primeiro elemento da lista, a próxima ação é novamente redução. A produção 4, $M \rightarrow P$, é a única aplicável e o novo estado das estruturas de dados é:

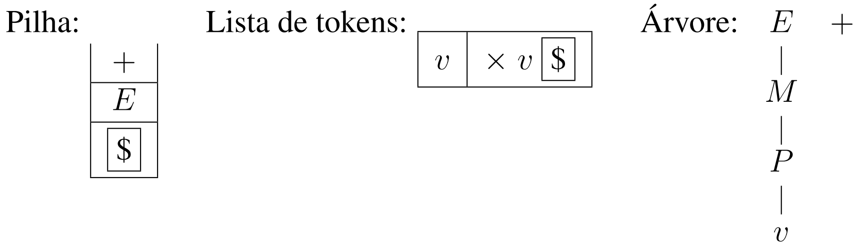
Pilha: M / $\boxed{\$}$ Lista de tokens: $+ \mid v \times v \boxed{\$}$ Árvore:

M
|
P
|
v

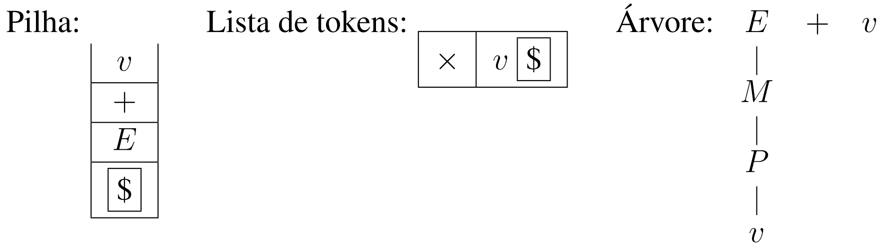
Para a entrada $M, +$ a ação é de redução. Novamente há apenas uma produção aplicável, a produção 2, $E \to M$:



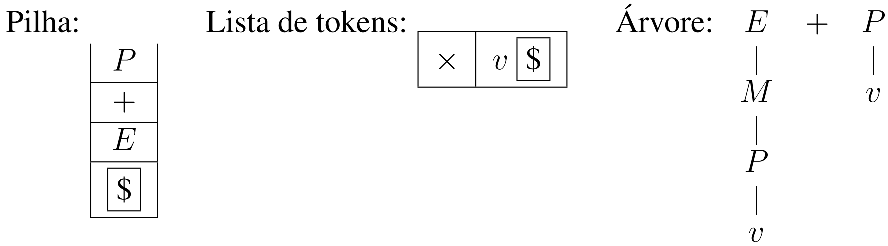
Para a entrada $E, +$ a ação é de deslocamento:



Novamente a ação é de deslocamento para a entrada $+, v$ da tabela:



A ação indicada na tabela DR para a entrada $v, \times$ é de redução. A única produção aplicável é a produção 6, $P \to v$:



Para esse estado das estruturas do analisador sintático, novamente a ação de redução é indicada, agora pela entrada $P, \times$ da tabela DR. A única produção cujo lado direito combina com os símbolos no topo da pilha é a produção 4, $M \to P$:

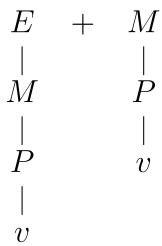
Pilha:

M
$+$
E
$

Lista de tokens:

$\times$	v $

Árvore:



Para a entrada $M, \times$ a ação indicada na tabela DR é de deslocamento. Portanto, o símbolo $\times$ é transferido da lista de tokens para o topo da pilha:

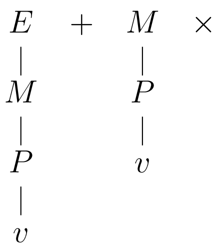
Pilha:

$\times$
M
$+$
E
$

Lista de tokens:

v	$

Árvore:



A próxima ação também é de deslocamento, como indicado na tabela DR para a entrada $\times, v$:

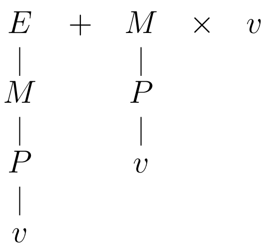
Pilha:

v
$\times$
M
$+$
E
$

Lista de tokens:

$

Árvore:



Para a entrada v, $, a ação indicada na tabela DR é de redução. A produção 6, $P \rightarrow v$, é a única aplicável aos símbolos no topo da pilha:

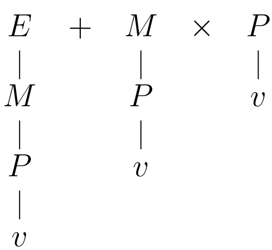
Pilha:

P
$\times$
M
$+$
E
$

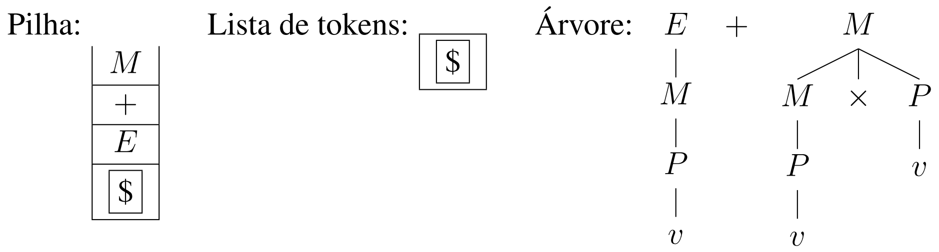
Lista de tokens:

$

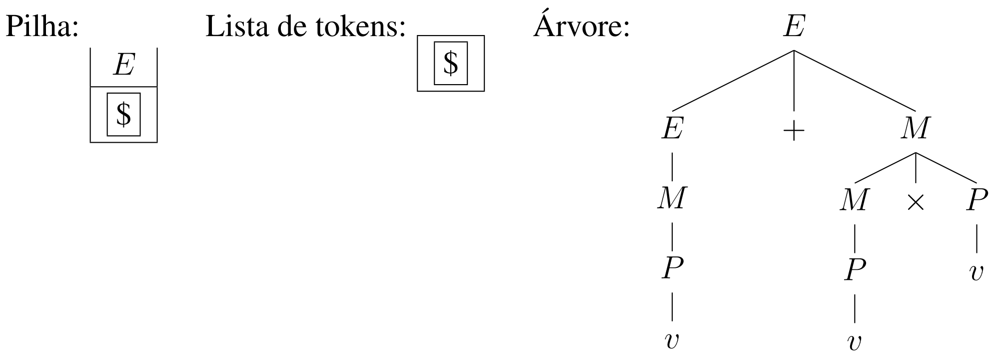
Árvore:



Com o topo da pilha P e o delimitador de sentença no início da lista de tokens, a ação indicada é de redução. Nessa situação, há duas produções que poderiam ser aplicadas aos símbolos no topo da pilha. Pela produção 3, os três símbolos $M \times P$ seriam substituídos pelo símbolo M. Se a produção 4 fosse aplicada, apenas o símbolo P seria substituído pelo símbolo M. A resolução desse conflito num analisador de precedência fraca é simples, pois pelas condições que a gramática atende não há nenhuma relação de Wirth-Weber entre o símbolo $\times$ e M, e a aplicação da produção 4 deixaria esses dois símbolos adjacentes, o que levaria a uma condição de erro adiante no procedimento. Portanto, a produção 3, $M \rightarrow M \times P$, deve ser a escolhida:



Nesse estado das estruturas do analisador, com M no topo da pilha e $ como próximo token, novamente a ação indicada na tabela DR é de redução. Mais uma vez há duas possíveis produções a aplicar, nesse caso a produção 1, $E \rightarrow E + M$, ou a produção 2, $E \rightarrow M$. Pelo mesmo raciocínio, o analisador seleciona a produção 1:



Como o único símbolo da gramática na pilha é o símbolo sentencial e a lista de tokens só contém o delimitador de sentença, o estado alcançado indica a validade da sentença, cuja estrutura é a indicada pela árvore sintática.

4.6 Geradores de analisadores sintáticos

Claramente, é possível construir analisadores sintáticos por meio da construção das tabelas que controlam a execução dos autômatos e da implementação dos respectivos procedimentos para varrer as seqüências de símbolos sob análise. No entanto, como ocorre na construção de analisadores léxicos, a construção de programas analisadores sintáticos é usualmente suportada por ferramentas para a geração automática de programas a partir de uma especificação. Uma tradicional ferramenta de criação de analisadores sintáticos é o `yacc` (*Yet Another Compiler-Compiler*), oriundo do ambiente de desenvolvimento de software que foi criado para o sistema operacional Unix. Assim como a ferramenta `lex`, o `yacc` tem por entrada um arquivo de especificação de uma gramática e gera como saída um módulo com código-fonte em C contendo uma rotina que realiza o reconhecimento de sentenças segundo essa gramática.

O analisador criado pelo `yacc` é um analisador de construção ascendente ou LR, o método de análise mais geral que pode ser aplicado a linguagens e gramáticas passíveis de análise determinística. Um analisador LR realiza a análise com a leitura dos símbolos terminais da esquerda para a direita (*left to right*) e obtém a derivação canônica mais à direita (*rightmost derivation*).

Uma gramática LR(k), usada como base de um analisador ascendente, é uma na qual as situações de conflito podem ser resolvidas pela verificação dos símbolos já lidos até o momento e pela visão de uma quantidade limitada a no máximo k símbolos adiante (o chamado *lookahead*). Na prática, o valor de k é geralmente limitado a 0 ou 1 sem perda de generalidade na aplicação do método. Embora haja gramáticas LR(2) que não são gramáticas LR(1), há um resultado teórico que diz que toda linguagem gerada por uma gramática LR(k) pode ser também gerada por uma gramática LR(1).

O programa gerado pelo `yacc` é um analisador sintático LR(1). Métodos de análise ascendente são quase sempre determinísticos, mas há situações, dependendo da gramática, em que o analisador deve decidir entre dois possíveis movimentos. Uma delas é a situação de conflito reduzir ou deslocar, e a outra, quando pelo menos duas regras são aplicáveis em uma situação de redução, é a situação de conflito reduzir ou reduzir. O programa gerado pelo `yacc` lida com algumas dessas situações, mas há outras nas quais um erro é indicado no processamento da especificação e o projetista precisa rever essa especificação.

O analisador sintático requer o auxílio de um analisador léxico para sua operação. Embora qualquer analisador possa ser utilizado, em geral o `yacc` utiliza o analisador léxico gerado pelo `lex` para esse fim.

4.6.1 Especificação da gramática

O núcleo da definição para a construção do analisador sintático em `yacc` é o conjunto de produções da gramática livre de contexto. O `yacc` utiliza uma notação baseada em BNF, que foi descrita na Seção 2.5.2, para a definição das regras da gramática, que são especificadas em um arquivo no formato texto.

O arquivo de entrada, que por convenção recebe a extensão `.y`, é estruturado em três seções. Como na definição de arquivos `lex`, essas três seções — definições, regras da gramática e código do usuário — são separadas pelos símbolos `%%`.

Cada produção na seção de regras é expressa na forma

```
simb : exp ;
```

Nesse caso, `simb` é um símbolo não-terminal e `exp` é a sua expansão em termos de outros símbolos da gramática. A expansão pode conter símbolos terminais e não-terminais, que por convenção são representados usando letras maiúsculas e minúsculas, respectivamente.

Pelas características de gramáticas livres de contexto, a expansão pode ser recursiva, isto é, conter o próprio símbolo que está sendo definido. Por exemplo, a primeira regra da gramática apresentada na Figura 4.2 pode ser expressa na notação de `yacc` como

```
expr : expr SOMA expr ;
```

Nessa expressão, `SOMA` é um nome simbólico utilizado para representar o token correspondente ao símbolo $+$.

Como na gramática, pelo menos uma expansão para esse símbolo deve ser não-recursiva. Por exemplo, para a última regra da gramática,

```
expr : VALOR ;
```

na qual `VALOR` é a representação de um token que foi reconhecido como um valor válido para a aplicação.

Em caso de haver na especificação definições recursivas, pelas características do analisador do tipo LR(1) que é gerado por `yacc`, recomenda-se optar pela recursão à esquerda.

Como em BNF, produções para um mesmo símbolo podem ser agrupadas usando o símbolo `|`,

```
expr    :   expr SOMA expr
        |   VALOR
        ;
```

Expansões para a string vazia também podem ser definidas. Nesse caso, a parte referente à expressão é deixada vazia. Por convenção e para tornar mais clara a definição, esse tipo de definição é destacado com um comentário em C:

```
retv    :   /* vazia */
        |   expr
        ;
```

Há duas formas de estabelecer qual símbolo não-terminal é o símbolo sentencial da gramática. A forma explícita é por meio da declaração `start` na seção de definições:

```
%start   simb
```

Caso a declaração `start` não esteja presente, o gerador assume implicitamente que o símbolo não-terminal da primeira produção da seção de regras é o símbolo sentencial da gramática.

Essa seção pode conter também definição dos nomes de tipos de tokens, os quais serão usados posteriormente nas expansões das produções. Tokens que são representados por um único caractere, como `'+'` ou `';'`, não precisam ser declarados e podem eventualmente ser usados dessa forma, como constantes do tipo caractere em C, nas expansões; os demais tokens precisam ser explicitamente declarados. Nesse caso, a declaração `token` é utilizada:

```
%token VALOR
```

Alternativamente, tokens para operadores podem ser definidos com uma especificação de associatividade usando, em vez de `token`, as declarações `left`, `right` ou `nonassoc`. Uma declaração

```
%left SOMA
```

determina que uma expressão `A + B + C` será interpretada como `(A + B)`

`+ C`, se o token `SOMA` está associado ao símbolo +. Por outro lado, se a declaração tivesse sido

```
%right SOMA
```

a interpretação para a mesma expressão seria `A + (B + C)`. No entanto, se o operador `SOMA` fosse definido com a declaração

```
%nonassoc SOMA
```

o programa indicaria que a expressão `A + B + C` é incorreta, pois o operador foi definido como não-associativo e portanto não poderia ser encadeado em uma expressão.

A precedência dos operadores também é definida com essas declarações. Operadores definidos na mesma linha de declaração, como

```
%left SOMA SUB
```

têm a mesma precedência. Para aqueles definidos em linhas distintas, as últimas declarações têm maior precedência. Por exemplo, para indicar que o operador `MULT`, para multiplicação, tem precedência maior que o operador `SOMA`, suas declarações devem estar em linhas distintas:

```
%left SOMA
%left MULT
```

O símbolo terminal `error` é um símbolo terminal predefinido em `yacc`. Este é utilizado como a última expansão de um símbolo quando o projetista da aplicação deseja determinar um curso de ação específico em uma situação de não-reconhecimento de uma sentença a partir das expansões previamente definidas para o símbolo.

Além das declarações próprias de `yacc`, a seção de declarações pode conter definições e declarações de variáveis na linguagem C, que podem ser utilizadas nos fragmentos de código associados às ações e também na seção de código do usuário. Da mesma forma que ocorre com os arquivos de especificação em `lex`, tais declarações e definições devem ocorrer entre os símbolos `%{` e `%}`.

4.6.2 Manipulação das sentenças reconhecidas

Reconhecer que uma seqüência de símbolos é uma sentença válida em uma gramática é parte essencial do processo de compilação, porém pouco uso teria se simplesmente uma indicação de validade fosse retornada sem nenhuma possibilidade de manipulação adicional das expressões. No caso de `yacc`, essa possibilidade está associada à definição de ações semânticas.

Uma ação semântica em `yacc` é definida como um bloco de expressões em C associado à definição de produções para um símbolo não-terminal:

```
simb  :  expansao    { acao }   ;
```

A definição do corpo da ação pode conter referências aos valores semânticos de cada um dos símbolos da produção. O valor semântico de um token está associado a um valor associado ao símbolo. Por exemplo, para um analisador sintático de um compilador para a linguagem de programação C que define tipos de tokens `IDENT` e `CONST` para representar identificadores e constantes, respectivamente, na linha de código C

```
int x;
```

a string `x` poderia ser reconhecida como um token do tipo `IDENT` com valor semântico 'x'. Similarmente, na linha

```
x = 10;
```

a string `10` poderia ser reconhecida como um token do tipo `CONST` com valor semântico 10 (inteiro).

O valor semântico do token pode ser referenciado na expressão C do arquivo `yacc` por meio de pseudovariáveis com nome $\$i$, onde i determina a posição do token na expansão. A variável `$$` referencia o valor semântico resultante, ou seja, para o símbolo do lado esquerdo da expressão. Por exemplo,

```
expr  :  expr SOMA expr
         { $$ = $1 + $3 }
      ;
```

atribui à expressão reduzida o valor semântico que é a soma dos valores semânticos do primeiro e do terceiro componentes da expansão, que estão separados pelo segundo componente, o token `SOMA`.

Se nenhuma ação for definida para uma produção, a ação semântica padrão, que é atribuir o valor do primeiro símbolo do lado direito ao símbolo do lado esquerdo — ou seja, `{ $$ = $1; }` —, é assumida.

O tipo associado a valores semânticos é definido pela macro `YYSTYPE`, ou seja, por uma definição para o pré-processador C que é substituída pela string com o seu tipo efetivo. Inicialmente, a definição de `YYSTYPE` é para o tipo `int`. Para modificar esse padrão, pode-se utilizar uma definição para o pré-processador C na primeira seção do arquivo que define a gramática, como por exemplo

```
%{
 #define   YYSTYPE  double
%}
```

Em aplicações que necessitem manipular tokens com diferentes tipos de valores semânticos, a declaração `union` deve ser utilizada para definir quais são os tipos de valores possíveis. Por exemplo, em uma aplicação que manipula valores inteiros e reais, a seguinte declaração estaria presente:

```
%union {
    int      ival;
    double  fval;
}
```

Tal declaração determina que a coleção de tipos de valores permitidos é composta por valores com nome `ival` ou `fval`, respectivamente, para valores inteiros e reais — no código C, uma estrutura com conteúdos alternativos (*union*) será criada. Esses mesmos nomes são utilizados para qualificar a definição de tokens da gramática, como em

```
%token <ival> INTEIRO
%token <fval> REAL
```

Quando uma coleção de tipos de valores é utilizada, é preciso determinar também qual o tipo para o símbolo não-terminal para o qual a expressão está sendo reduzida. Para esse fim, `yacc` define a declaração `type`:

```
%type <fval> expr
```

Pelas definições anteriores, essa declaração estabelece que o valor semântico do símbolo não-terminal `expr` será do tipo `double`, que está associado a `fval`.

4.6.3 Desenvolvimento de uma aplicação

Esta seção apresenta o procedimento para desenvolver uma aplicação que realiza a análise sintática de um arquivo de entrada usando a ferramenta `bison`. Essa ferramenta é uma implementação de `yacc` disponível para diversas plataformas que é distribuída, assim como `flex`, sob a licença de software GNU da Free Software Foundation.

A execução de `bison` requer como argumento o nome do arquivo com a gramática especificada. Se esse arquivo recebeu, por exemplo, o nome `mygram.y`, então a linha de comando

```
bison mygram.y
```

gera um arquivo com código-fonte em C de nome `mygram.tab.c`. Esse arquivo contém a definição das tabelas para a análise sintática e a rotina de reconhecimento, de nome é `yyparse()`, que pode ser integrada a outras aplicações.

A rotina `yyparse()` deve ser invocada pela aplicação para que a análise sintática do arquivo de entrada seja realizada. Essa rotina retorna um valor inteiro, que será 0, se toda a entrada pode ser reconhecida sem erros pela gramática especificada, ou 1, caso algum erro sintático tenha sido detectado no arquivo de entrada.

A rotina `yyparse()` invocará uma rotina `yylex()` que irá varrer o arquivo de entrada e retornar os tokens para o analisador sintático. Essa rotina não é criada automaticamente e deve ser fornecida pelo usuário. Tipicamente, mas não necessariamente, é gerada por uma ferramenta `lex`.

Caso a rotina `yylex()` fornecida seja simples o suficiente para ser definida manualmente, então seu código pode ser incluído na seção de usuário do arquivo de especificação da gramática e todas as definições de tipos de tokens podem ser usadas diretamente. Caso contrário — por exemplo, se `flex` for utilizada para gerar a rotina de análise léxica —, é preciso transportar essas definições para os demais módulos da aplicação. Para tanto, utiliza-se a chave de execução `-d`, que gera um arquivo de cabeçalho com essas definições:

```
bison -d mygram.y
```

Com essa opção, além do arquivo-fonte C, um arquivo de nome `mygram.tab.h` é gerado com as definições necessárias, podendo ser incluído em outros módulos para realizar a integração. Por exemplo, um arquivo `mylex.l` contendo a especificação para reconhecimento de tokens usando `flex` poderia conter, na sua seção de definições, a declaração

```
%{
 #include "mygram.tab.h"
%}
%% /* definicoes das expressoes regulares: */
```

Outro aspecto importante na integração das rotinas `yyparse()` e `yylex()` é na forma de definição dos valores semânticos e dos tipos dos tokens. A definição do tipo de token é determinada pelo valor de retorno de `yylex()`. Por exemplo, se no arquivo `mygram.y` houvesse a declaração

```
%token VALOR
```

então a ação associada ao reconhecimento de um padrão regular que reconhecesse esse tipo de token deveria fechar com a expressão

```
return VALOR;
```

O outro ponto de ligação entre as duas rotinas é a definição do valor semântico, que é realizado por meio de uma variável global `yylval`, definida no módulo `mygram.tab.c`. Para aplicações que trabalham com um único tipo de valor, basta atribuir na ação associada ao reconhecimento do token o valor semântico resultante, como em

```
...
yylval = atoi(yytext);
return VALOR;
}
```

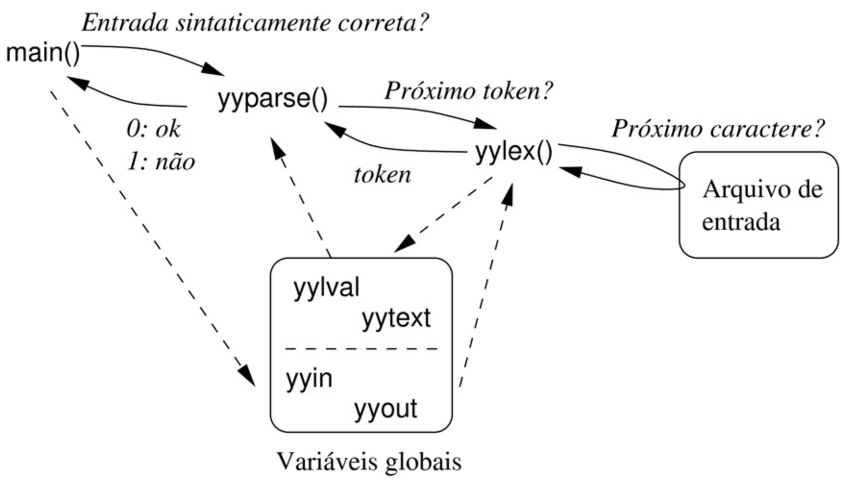
Se a aplicação opera com múltiplos tipos de valores semânticos, então será preciso qualificar a atribuição, indicando qual componente da união de tipos está sendo utilizado:

```
...
yylval.ival = atoi(yytext);
return VALOR;
}
```

A Figura 4.10 ilustra como é feita a integração das rotinas produzidas pelo gerador de analisador sintático, `yyparse`, e pelo gerador de analisador léxico,

yylex. A partir do código principal da aplicação, a rotina yyparse é invocada. Eventualmente, esse código pode alterar os valores das variáveis globais yyin e yyout, definidas no módulo do analisador léxico, para determinar quais arquivos serão utilizados para a entrada e a saída de yylex. A rotina yyparse irá solicitar da rotina yylex que obtenha os tokens do arquivo de entrada; para tanto, yylex lê os caracteres desse arquivo, compondo os tokens segundo as expressões regulares definidas. Uma vez reconhecido um token, yylex atribui seu valor semântico por meio da variável global yylval, definida no módulo do analisador sintático, e qual a string correspondente com a variável global yytext, definida pelo analisador léxico.

Figura 4.10 Operação integrada dos analisadores léxicos e sintáticos



Exemplo

Esta seção ilustra como a ferramenta bison é utilizada para criar uma aplicação baseada na gramática de expressões G_2 da Figura 4.2. Nessa aplicação, identificadores são substituídos por constantes de valor inteiro nas expressões. A descrição dessa aplicação, que pode ser usada como base para uma calculadora simples, serve para ilustrar detalhes da integração dos analisadores léxico e sintático em C++, além de alguns aspectos do tratamento da entrada e de erros nesse tipo de aplicação.

A parte essencial da especificação da gramática é o conjunto de regras para expressões, equivalentes às produções da gramática. Com a adoção da conven-

ção usada em arquivos de especificação `yacc`, de utilizar símbolos em minúsculas para símbolos não-terminais e em maiúsculas para símbolos terminais, essas regras são representadas como

```
expr    :    expr SOMA expr            {  $$ = $1 + $3; }
        |    expr MULT expr            {  $$ = $1 * $3; }
        |    ABRPAR expr FECPAR        {  $$ = $2;      }
        |    VALOR
        ;
```

O conjunto de símbolos terminais utilizados nessas expressões é definido na seção inicial do arquivo:

```
%token  VALOR
%left   SOMA
%left   MULT
%token  ABRPAR FECPAR
```

Essa seqüência de definições elimina a ambigüidade da gramática ao estabelecer precedência diferente para a operação de soma e de multiplicação.

Esses tokens são reconhecidos pelo analisador léxico; o arquivo de especificação para o `lex` contém as regras correspondentes:

```
[0−9]+  {
        yylval = atoi(yytext);
        return VALOR;
        }
\+      { return SOMA; }
\*      { return MULT; }
\(      { return ABRPAR; }
\)      { return FECPAR; }
```

Para que os símbolos utilizados sejam reconhecidos pelo analisador léxico, é preciso repassar as definições de tokens feitas no arquivo de especificação do analisador sintático, no padrão `yacc`, para o arquivo de especificação do analisador léxico, no padrão `lex`. A maneira de realizar essa conexão entre os dois arquivos é por meio da criação de um arquivo de cabeçalho no padrão C durante o processamento da especificação do analisador sintático; esse arquivo é então incluído com uma diretiva para o pré-processador C no arquivo de especificação do analisador léxico. Por exemplo, se o arquivo de especificação do analisador sintático tem o nome `espec1.y`, a invocação do comando gerador de analisador sintático com a opção `-d`, como em

```
> bison -d espec1.y
```

cria, além do arquivo com o código C para o analisador sintático, um arquivo de cabeçalho `espec1.tab.h` com as definições dos tokens. Esse arquivo pode ser incluído na especificação do analisador léxico com uma diretiva

```
%{
#include "espec1.tab.h"
...
%}
```

Assim, os mesmos nomes simbólicos para os tokens que foram definidos com as diretivas `token`, `left`, `right` e `nonassoc` podem ser utilizados nas definições da especificação do analisador léxico.

Além das regras sintáticas para a gramática de expressões, outras regras são necessárias para processar a entrada e apresentar o resultado da expressão. Para apresentar o resultado, um novo símbolo não-terminal é introduzido:

```
result  :   FIMLIN
        |   expr FIMLIN    {  cout << "Resposta: " << $1 << endl; }
        |   error FIMLIN   {  yyerrok;   }
        ;
```

Três regras são associadas ao símbolo. A primeira é a entrada para que entradas com linhas em branco sejam simplesmente ignoradas, haja vista que não há nenhuma ação associada ao reconhecimento dessa regra — `ENDLINE` é outro token definido pelo analisador sintático e que também deve ser reconhecido pelo analisador léxico. A segunda é a que reconhece uma expressão válida em uma linha e, na ação associada, apresenta o valor associado à expressão. A terceira regra utiliza o símbolo predefinido `error` e, na ação associada, a macro `yyerrok`, que desconsidera os eventuais erros na entrada e permite a continuidade da operação do analisador.

Outro aspecto associado ao tratamento de erros é a apresentação da mensagem de erro. A rotina do analisador sintático invoca, em caso de erro, a rotina `yyerror`, que deve ser definida pelo programador. Por exemplo, uma rotina simples para apresentar uma mensagem que indique a condição de erro e apresente qual foi a última string processada na entrada, que levou a essa condição de erro, é apresentada a seguir.

```
void yyerror(char* msg) {
    extern char* yytext;
```

```
    cout << msg << ": " << yytext << endl;
}
```

Finalmente, um símbolo `entrada` é introduzido na especificação para permitir que o processamento continue por várias linhas e não seja interrompido após uma única linha:

```
entrada : /* vazia */
        | entrada result
        ;
```

Duas regras são associadas ao símbolo `entrada`. A primeira determina que uma entrada vazia — a situação que ocorre quando as teclas `control` e `d` (control-d) são pressionadas simultaneamente no início de uma linha — é também válida, mesmo que não gere nenhuma ação nesse caso. Na realidade, tal entrada é reconhecida como o fim do arquivo de entrada e, caso não seja indicada uma ação em contrário, encerra o processamento. A segunda regra, recursiva à esquerda, permite interpretar várias linhas com expressões como uma entrada válida.

É interessante aproveitar esse exemplo para mostrar como é possível desenvolver uma aplicação em C++ que utilize o código gerado automaticamente para os dois analisadores. Como os geradores de analisador léxico e sintático são anteriores ao surgimento de C++ e produzem código em C, há alguns aspectos adicionais que devem ser considerados para essa integração. A forma mais simples de fazer essa conexão entre esses dois analisadores é indicar, para o código de aplicação do analisador sintático que pode estar em C++, que as funções produzidas pelos geradores de analisadores são funções C. Para isso, a indicação `extern "C"` é utilizada:

```
    extern "C"
    {
        int yyparse(void);
        int yylex(void);
        int yywrap()
        {
            return 1;
        }

    }
```

O papel dessa indicação ficará claro quando forem apresentadas as tabelas de símbolos, na Seção 5.1.

Agora já é possível apresentar o arquivo completo da especificação do analisador sintático para uma aplicação desenvolvida em C++:

```
%{
// Semente para calculadora
#include <iostream>
using namespace std;
    extern "C"
    {
            int yyparse(void);
            int yylex(void);
            int yywrap()
            {
                    return 1;
            }

    }
void yyerror(char *);
%}
%start entrada
%token  VALOR FIMLIN
%left   SOMA
%left   MULT
%token  ABRPAR FECPAR
%%
entrada :       /* vazia */
        |       entrada result
        ;
result  :       FIMLIN
        |       expr FIMLIN     {  cout << "Resposta: " << $1 << endl; }
        |       error FIMLIN    {  yyerrok;   }
        ;
expr    :       expr SOMA expr {  $$ = $1 + $3; }
        |       expr MULT expr {  $$ = $1 * $3; }
        |       ABRPAR expr FECPAR   {  $$ = $2;     }
        |       VALOR
        ;
%%
void yyerror(char* msg) {
   extern char* yytext;
   cout << msg << ": " << yytext << endl;
}
```

Para processar o arquivo, o comando `bison` é utilizado. Se o arquivo recebeu o nome `acumuly.y`, então a linha de comando para realizar esse processamento é:

```
>  bison -d acumuly.y
```

Com esse comando, dois arquivos são produzidos: o código da aplicação e do analisador sintático, no arquivo `acumuly.tab.c`, e o arquivo com definições para os símbolos terminais, no arquivo `acumuly.tab.h`. Este é utilizado no arquivo com as especificações para o analisador léxico:

```
%{
#include "acumuly.tab.h"
extern YYSTYPE yylval;
%}
%%
[0-9]+          {
                yylval = atoi(yytext);
                return VALOR;
                }
\+              { return SOMA; }
\*              { return MULT; }
\(              { return ABRPAR; }
\)              { return FECPAR; }
\n              { return FIMLIN; }
%%
```

Se esse arquivo recebe o nome de `acumuly.l`, então pode ser processado com o comando `flex`:

```
> flex acumuly.l
```

Com esse comando, o arquivo com o código-fonte para o analisador léxico, com o nome `lex.yy.c`, é criado.

Nesse momento, há dois arquivos com código C produzido pelos dois geradores e que precisam ser compilados para que sejam integrados ao código da aplicação. O primeiro módulo a ser compilado é o arquivo que contém o código do analisador léxico:

```
> gcc -c lex.yy.c
```

A chave `-c` para o compilador sinaliza que o arquivo indicado não é uma aplicação completa e o módulo objeto gerado será posteriormente integrado a outro

módulo. O módulo objeto será armazenado em um arquivo `lex.yy.o`, gerado por esse comando. Esse arquivo pode ser combinado com a aplicação e o analisador sintático durante a compilação do segundo arquivo, que contém o código produzido pelo gerador de analisador sintático e, nesse caso, também o código da aplicação que foi especificado na seção de código de usuário do arquivo `acumuly.y`. Como nesse arquivo foi introduzido código C++, o processamento deve ser realizado com compilador para essa linguagem:

```
> g++ -o acumuly lex.yy.o acumuly.tab.c -ly -lfl
```

Nessa linha de comando, as bibliotecas para `bison` e para `flex` foram especificadas com a chave `-l`. A chave `-o` indica qual o nome que será utilizado para o arquivo com o código executável — nesse caso, o nome escolhido foi `acumuly`.

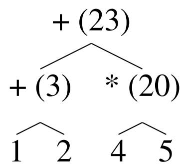
Um exemplo de execução da aplicação assim gerada é reproduzido a seguir. Na linha de comando, o usuário pode invocar o programa executável que foi produzido:

```
> ./acumuly
```

O programa, ao executar, espera que o usuário digite uma expressão a ser processada. Por exemplo,

```
1+2+4*5
```

terminada com a digitação da tecla `enter`. Nesse caso, a aplicação irá reconhecer a sentença e associar a ela a estrutura



na qual o valor entre parênteses é o valor semântico da expressão binária que foi atribuído pelo analisador. Ao reconhecer essa expressão como válida e ao combiná-la com o token de fim de linha, a saída para o usuário é produzida:

```
Resposta: 23
```

Na seqüência, o usuário pode digitar outras entradas válidas que o analisador as reconhece e produz a saída correspondente, como

```
2+2
Resposta: 4
```

Caso o usuário digite uma entrada contendo erros, o analisador deve indicar a situação com a apresentação da mensagem de erro padrão:

```
1=2+3+4
=syntax error: 2
```

Tal situação leva à invocação da macro `yyerrok`, de modo que o erro nessa linha não impede que novas linhas sejam processadas corretamente, como

```
1+2
Resposta: 3
[^d]
>
```

Nesse caso, após a última resposta o usuário digitou as teclas `control` e `d`, e assim encerrou a entrada de expressões. Com isso, a aplicação foi encerrada e o sistema operacional está pronto para receber novos comandos.

4.7 Exercícios

4.1 Mostre, para a sentença 00001111 na gramática G_1 (Seção 2.3), a sua árvore sintática, as derivações canônicas mais à esquerda e mais à direita e as correspondentes seqüências de reconhecimento.

4.2 Mostre a árvore sintática, as derivações canônicas e as correspondentes seqüências de reconhecimento para a sentença $aabb$ na gramática $G_b = \{V_t, V_n, P, S\}$, com $V_t = \{a, b\}$, $V_n = \{A, S\}$ e as produções $P = \{S \rightarrow A,\ A \rightarrow aAb,\ A \rightarrow ab\}$.

4.3 Mostre a árvore sintática, as derivações canônicas e as correspondentes seqüências de reconhecimento para a sentença $x\,x\,x\,y\,y\,z\,x\,x\,z$ na gramática G_c, com $V_n = \{S, A, B, C\}$, $V_t = \{x, y, z\}$, símbolo sentencial S e produções $S \rightarrow AxByC$, $A \rightarrow xAx$, $A \rightarrow \varepsilon$, $B \rightarrow By$, $B \rightarrow \varepsilon$, $C \rightarrow zAz$.

4.4 Dada a gramática com símbolo não-terminal e sentencial S, símbolos terminais a, b e produções

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

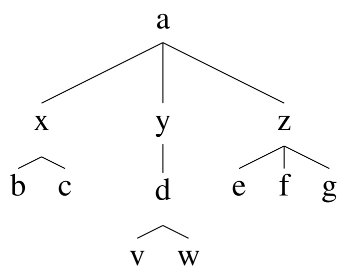
mostre, usando a sentença *abab*, que esta é ambígua. Para tanto, apresente para a sentença:

(a) Duas derivações canônicas mais à direita;

(b) Duas derivações canônicas mais à esquerda;

(c) Duas árvores sintáticas.

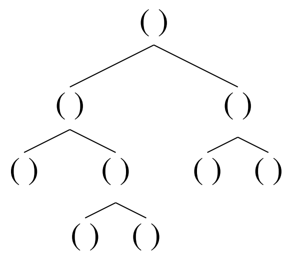
4.5 Para a árvore



(a) Qual é a seqüência de apresentação dos seus elementos quando a estratégia de varredura pré-ordem é adotada?

(b) Qual é a seqüência de apresentação dos seus elementos quando a estratégia de varredura pós-ordem é adotada?

4.6 Uma árvore binária com nove nós apresenta a seguinte estrutura:



Atribua a cada nó, preservando essa estrutura, os dígitos de 1 a 9 de forma que a varredura da árvore apresente os valores em ordem ascendente quando a estratégia adotada é:

(a) pré-ordem;

(b) intra-ordem;

(c) pós-ordem.

4.7 Apresente todas as árvores sintáticas para as seguintes sentenças em G_2 (Seção 4.1):

(a) $v \times v$

(b) $v \times v + v$

(c) $v \times (v + v)$

(d) $v + v \times v + v$

(e) $(v + v) \times (v + v)$

4.8 Considere a gramática G_g com símbolos não-terminais $\{S, L\}$, símbolo sentencial S, símbolos terminais $\{(,), a, \triangle\}$ e produções

$$
\begin{array}{lcl}
S & \rightarrow & (L) \\
S & \rightarrow & a \\
L & \rightarrow & L \triangle S \\
L & \rightarrow & S
\end{array}
$$

(a) Apresente as árvores sintáticas para as sentenças:

- $(a \triangle a)$
- $(a \triangle (a \triangle a))$
- $(a \triangle ((a \triangle a) \triangle (a \triangle a)))$

(b) Monte a tabela sintática para um analisador sintático preditivo para reconhecer sentenças nessa linguagem e mostre a operação do analisador sintático preditivo para cada uma das sentenças do item *a*.

(c) Obtenha as relações de Wirth-Weber entre os símbolos da gramática G_g.

(d) Se possível, construa o analisador de precedência fraca para a gramática G_g e mostre sua operação para cada uma das sentenças do item *a*. Caso contrário, indique qual condição é violada.

4.9 Para a gramática $G_h = \{V_n, V_t, P, S\}$, com $V_n = \{A, S\}$, $V_t = \{a, b\}$ e produções $P = \{S \rightarrow A,\ A \rightarrow aAb,\ A \rightarrow ab\}$.

(a) Monte a tabela sintática para o analisador preditivo, se possível.

(b) Obtenha as relações de Wirth-Weber e mostre que a gramática é de precedência fraca.

(c) Monte a tabela DR para o analisador de precedência fraca.

(d) Mostre a operação do analisador no reconhecimento da sentença $aabbb$, com a correspondente construção da árvore sintática a cada passo.

4.10 Considere a gramática G_i cujas produções são apresentadas a seguir, com símbolo sentencial S e símbolos terminais $\{a, e, o, l, x\}$:

$$
\begin{array}{rcl}
S & \rightarrow & ABe \\
A & \rightarrow & a \\
A & \rightarrow & o \\
B & \rightarrow & x \\
B & \rightarrow & l
\end{array}
$$

(a) Construa o analisador sintático preditivo.

(b) Construa o analisador sintático de precedência fraca.

(c) Mostre a operação dos dois analisadores no reconhecimento da sentença axe.

4.11 O seguinte fragmento de um arquivo `yacc` define os símbolos e produções para uma gramática G_j:

```
%token  T1, T2, T3, T4
%%
n1   :  n2 T1 n2 T2  |  T2 ;
n2   :  n3 |  n4 ;
n3   :  n4 T3 n4 |  T3 n4 ;
n4   :  n4 T4 |  T4 ;
```

(a) Mostre a representação formal para G_j.

(b) Mostre a representação de G_j em BNF.

(c) Mostre a representação gráfica de G_j na notação de diagramas sintáticos.

4.12 O seguinte fragmento de uma especificação `yacc` tenta representar o comando condicional (`IF`) de alguma linguagem:

```
%token CMD COND ELSE IF
%%
  stmt : CMD  |  ifstmt  ;
ifstmt : IF '(' COND ')' stmt
       | IF '(' COND ')' stmt ELSE stmt  ;
```

Ao compilar essa especificação, o gerador de analisador sintático produz uma mensagem de aviso para indicar que a gramática especificada tem uma ambigüidade que produz um conflito de deslocamento ou redução.

(a) Qual é a causa desse conflito na especificação?

(b) Complemente a especificação de modo que seja possível executar um exemplo que permita descobrir qual é a ação padrão tomada pelo analisador gerado numa dessas situações de conflito.

4.13 Complemente o exemplo da Seção 4.6.3 de forma a contemplar as quatro operações básicas e a incluir valores negativos com o operador unário −.

4.14 Complemente o exemplo da Seção 4.6.3 de forma a aceitar valores reais além de inteiros.

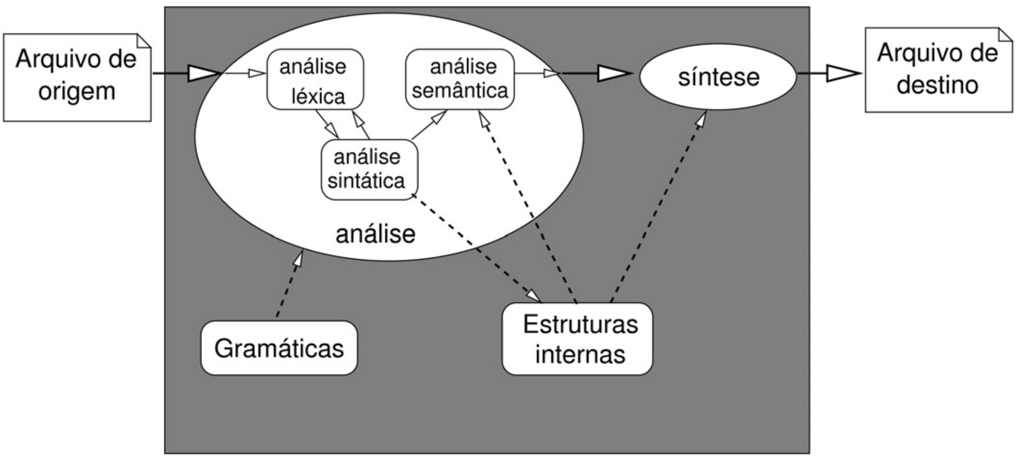
4.15 Complemente o exemplo da Seção 4.6.3 de forma a definir uma calculadora científica que aceita valores inteiros e reais e que, além das quatro operações aritméticas e de valores negativos, reconheça as operações da biblioteca matemática de C, especificadas no arquivo `cmath`, como as funções `pow`, `log` e `sin`.

Capítulo 5

Análise semântica

Os capítulos anteriores mostraram como o código-fonte de um programa é analisado para permitir o reconhecimento de elementos e de estruturas válidas. Embora a análise sintática consiga verificar se uma expressão obedece às regras de formação de uma dada gramática, seria muito difícil expressar por meio de gramáticas livres de contexto algumas regras usuais em linguagem de programação, como "todas as variáveis devem ser declaradas" e situações nas quais o contexto em que ocorre a expressão ou o tipo da variável deve ser verificado.

Figura 5.1 Atividades do compilador: análise semântica



O objetivo da análise semântica, atividade do compilador destacada na Figura 5.1, é trabalhar nesse nível de inter-relacionamento entre partes distintas do

programa. Essa fase utiliza as informações geradas na análise sintática, como a árvore sintática, e outras estruturas auxiliares que podem ser criadas durante a análise sintática, como a tabela de símbolos, para verificar a validade dessas regras não representadas pelas gramáticas livres de contexto.

5.1 Tabela de símbolos

A tabela de símbolos é uma estrutura auxiliar criada pelo compilador para apoiar as atividades da análise semântica do código. Um exemplo prático da sua utilidade é ilustrado pelo seguinte programa em C++:

```
#include <iostream>
using namespace std;

int main() {
        a = 10;
        cout << "Valor de a: " << a << endl;
}
```

Obviamente, cada instrução isoladamente está sintaticamente correta. O compilador consegue construir, para cada comando, uma árvore sintática correta. No entanto, a compilação desse programa apresenta a seguinte informação de erro:

```
...cpp: In function 'int main()':
...cpp:5: error: 'a' was not declared in this scope
```

Ou seja, na primeira utilização que se tenta fazer da variável `a`, o compilador reconhece que tal variável não foi declarada. Para poder detectar esse tipo de erro, ele precisa manter internamente a informação sobre quais variáveis já foram declaradas e podem ser utilizadas. Tal informação é mantida na tabela de símbolos.

No entanto, não basta manter apenas a informação de se uma variável foi ou não declarada. Considere o seguinte programa:

```
#include <iostream>
using namespace std;

int main() {
        int a = 9;
        float b = 5;
```

```
    cout << "a%b: " << a%b << endl;
}
```

A compilação desse arquivo também indicará erros:

```
...cpp: In function 'int main()':
...cpp:7: error: invalid operands of types 'int' and
     'float' to binary 'operator%'
```

Nesse caso, as duas variáveis envolvidas foram declaradas. Por tal motivo, a mensagem não é sobre variáveis não declaradas, mas sim sobre o fato de que os tipos para os operandos para o operador de módulo (o resto da divisão inteira) são inválidos. De fato, tal operador só admite operadores inteiros, mas `b` é do tipo real (float). Assim, é possível inferir que também a informação sobre o tipo de cada variável é mantida na tabela de símbolos.

Além da informação sobre o tipo de cada variável declarada, há outra informação sobre cada variável que é importante para o compilador e que deve ser mantida na tabela de símbolos. Considere o seguinte exemplo:

```
#include <iostream>
using namespace std;

int main() {
    int a = 9;
    void mostra();
    mostra();
}

void mostra() {
    cout << "a: " << a << endl;
}
```

Nesse caso, o erro indicado pelo compilador é:

```
...cpp: In function 'void mostra()':
...cpp:11: error: 'a' was not declared in this scope
```

Embora a mensagem seja parecida com aquela do primeiro exemplo, nesse caso existe uma variável `a` na tabela de símbolos, pois ela foi declarada na função `main`. No entanto, ao tentar usá-la na função `mostra`, o compilador reconheceu que para esta a variável não era válida, pois `a` é uma variável local a

`main` e portanto só pode ser utilizada no corpo dessa função. Esse tipo de informação sobre o escopo das variáveis declaradas também é mantido na tabela de símbolos.

Todos os identificadores utilizados em um programa, e não apenas variáveis, devem estar presentes na tabela de símbolos. Outra informação que deve estar presente na tabela de símbolos é, portanto, um atributo que indique o tipo de identificador ao qual aquele nome está associado — se é uma variável, um nome de função, um nome de estrutura ou de classe, uma palavra reservada da linguagem.

Nas linguagens de programação mais modernas, é possível que um mesmo nome de símbolo seja utilizado mais de uma vez. Linguagens como C++ permitem, por meio do mecanismo de sobrecarga (*overload*), que funções com o mesmo nome coexistam desde que os tipos de seus argumentos sejam diferentes. O conceito de espaço de nomes (*namespace*) também é um mecanismo que livra o programador da obrigação de criar nomes diferentes para seus identificadores, o que pode se tornar rapidamente um problema em aplicações com alguma complexidade.

O mecanismo que os compiladores adotam para lidar com tais situações é a decoração de nomes (*name mangling*). Em termos simples, o compilador cria um “nome interno” para cada identificador que combina o espaço no qual está declarado e, no caso de funções, qual o tipo de seus argumentos. Assim, quando uma função for invocada, o compilador inspeciona a tabela de símbolos não apenas pelo nome mas também pelos tipos de seus argumentos, de forma a obter o nome decorado correto para aquela invocação.

Por exemplo, a função `mostra` do exemplo anterior não tem argumentos e por isso o tipo de argumentos é reconhecido pelo compilador como `void` (vazio). Um compilador hipotético poderia criar um nome interno `H_mostra_v` para esse símbolo. Se uma outra função com o mesmo nome fosse criada mas recebesse um argumento do tipo inteiro, isso não causaria um conflito na tabela de símbolos pois seu nome interno seria `H_mostra_i`.

A estratégia para a decoração de nomes ainda é realizada de forma não padronizada e cada projetista de compilador adota sua maneira. Isso torna mais difícil a integração de módulos gerados por diferentes compiladores, mesmo que a linguagem de programação utilizada tenha sido a mesma. Em alguns casos, há mecanismos para sinalizar o compilador que alguns símbolos não devem seguir a sua estratégia de decoração, pois têm origem em módulos criados com outro compilador. É o caso em C++ quando se deseja integrar símbolos originários de uma biblioteca com rotinas desenvolvidas em C, quando a indicação

`extern "C"` é usada, como foi visto em um exemplo no capítulo anterior:

```
extern "C"
{
        int yyparse(void);
        int yylex(void);
        int yywrap()
        {
                return 1;
        }

}
```

Alguns aspectos relativos à implementação de tabelas por meio das classes parametrizadas de C++ já foram abordados na Seção 3.3.2. Como a chave para ter acesso às informações na tabela é o nome do símbolo, a estrutura de STL adequada para essa implementação é `map`. Cabem no entanto algumas observações adicionais nesse caso.

A primeira observação é que um objeto da classe parametrizada `map` não admite duas ou mais entradas associadas à mesma chave. Assim, se há duas variáveis de mesmo nome em escopos diferentes, apenas uma entrada será criada. Uma alternativa para lidar com esse problema é permitir mais de um valor associado a uma chave, ou seja, a entrada associada a uma chave não seria um único registro mas sim uma outra estrutura de dados, como uma lista, por exemplo, com uma entrada para cada uso daquele identificador. Outra alternativa é utilizar a estrutura `multimap`, que tem a mesma função de estruturas do tipo `map` mas que permite chaves duplicadas.

Outra observação é relativa à implementação da classe `map` na biblioteca STL de C++. A implementação padrão adota uma estrutura interna em árvore, o que garante tempo de acesso a um elemento da coleção com complexidade temporal logarítmica. Na construção de tabelas de símbolos para compiladores, é usual utilizar uma estrutura de tabela *hash*, com complexidade temporal constante para acesso a um elemento. Embora ainda não seja padronizado, é usual que implementações de STL apresentem alguma implementação de `map` usando essa estratégia, como `hash_map` no caso do compilador `g++`.

A última observação é sobre a manutenção da informação de escopo. Para tornar mais clara qual é a dificuldade, considere o seguinte fragmento de código:

```
int a, b;
... // escopo 1
```

```
void f() {
    float a, c;
    ... // escopo 2
}
... // escopo 1
void g() {
    int c, d;
... // escopo 3
}
```

Na parte do código marcada "escopo 1" há duas variáveis inteiras declaradas, `a` e `b`, além das funções `f` e `g`. Já na corpo da função `f` (escopo 2), há uma variável adicional do tipo real, `c`, e a variável `a` do escopo 1 é sobreposta pela variável real `a` declarada nesse escopo — a variável `b` de escopo 1 ainda é válida, no entanto. Já no corpo da função `g` (escopo 3), as duas variáveis do escopo 1 são visíveis e podem ser referenciadas, assim como as duas variáveis locais da função, `c` e `d`, com a ressalva de que a primeira delas é diferente da variável de mesmo nome que era local à função `f`.

Há duas formas básicas de manter a informação sobre o escopo nas tabelas de símbolos. A primeira é trabalhar com múltiplas tabelas, uma para cada escopo. Assim, quando o compilador realiza a análise semântica sobre o uso dos identificadores, ele utiliza como referência a tabela de símbolos relativa àquele escopo. A outra maneira é organizar a tabela de símbolos como uma pilha de tabelas, de maneira que a cada nova definição de escopo um conjunto de símbolos é agregado à pilha e, ao final do escopo, esse conjunto é desempilhado.

A tabela de símbolos é essencial para permitir a verificação da correção do programa na análise semântica, mas também tem outros usos. Na atividade de depuração, por exemplo, o programador não lida com nomes internos de variáveis, mas precisa reconhecer os identificadores que ele definiu. A informação mantida na tabela de símbolos permite fazer esse mapeamento.

5.2 Heurísticas para a análise semântica

Com a tabela de símbolos à disposição, o compilador pode realizar, por meio da análise semântica, a verificação de que o uso dos identificadores está de acordo com sua definição. Essa verificação realizada no código é denominada verificação estática e permite detectar algumas situações de erro, que podem ser descobertas apenas pela análise do código-fonte. Deve-se ressaltar, entretanto, que há outras situações de uso inadequado que só podem ser detectadas durante a

execução do programa, como a atribuição de um valor a uma variável além do limite representável por seu tipo ou uma divisão inteira pelo valor 0. Tais situações são apenas detectadas com uma verificação dinâmica de tipos, ou seja, realizada durante a execução do programa.

A análise semântica é realizada por meio de heurísticas, sem o mesmo grau de formalismo associado às análises léxica e sintática. O motivo para a adoção dessa estratégia é que a representação de qualquer regra associada a uma definição de um símbolo, seguida por uma quantidade qualquer de outros símbolos e por um posterior uso daquele primeiro símbolo, demandaria a representação por meio de gramáticas sensíveis ao contexto, para as quais não há mecanismos adequados de processamento automático.

As tarefas básicas desempenhadas durante a análise semântica incluem a verificação de tipos, a verificação do fluxo de controle e a verificação da unicidade da declaração de variáveis. Dependendo da linguagem de programação, outros tipos de verificações podem ser necessários.

A verificação de tipos tem por objetivo verificar se as variáveis em uma determinada operação estão coerentes com os tipos das variáveis que se espera para a operação. Um exemplo de uma situação inadequada foi visto na seção anterior, na aplicação do operador de módulo `%` a uma variável do tipo real. Um outro exemplo de detecção de erro com essa análise estática pode ser observado na tentativa de inicializar uma variável do tipo ponteiro, própria para endereços de variáveis, a partir de um valor inteiro:

```
int main() {
        int a = 0xFF0;
        int* b;

        b = a;
}
```

Nesse caso, o compilador detecta o problema e emite a seguinte mensagem de erro:

```
...cpp: In function 'int main()':
...cpp:8: error: invalid conversion from 'int' to 'int*'
```

ou seja, na linha 8 há uma tentativa inválida de conversão de uma variável escalar, do tipo `int`, para uma variável ponteiro, do tipo `int*`.

Em alguns casos, o compilador realiza a conversão automática de um tipo para outro que seja adequado à aplicação do operador. Por exemplo, na expressão em C

```
a = x - '0';
```

a constante do tipo caractere `'0'` é automaticamente convertida para inteiro para compor corretamente a expressão aritmética na qual ela toma parte; todo `char` em uma expressão é convertido pelo compilador para um `int`. Esse procedimento de conversão de tipo é denominado coerção (*cast*). Em C e C++, a seguinte seqüência de regras determina a realização automática de coerção em expressões aritméticas com dois operandos:

1. `char` e `short` são convertidos para `int`, `float` para `double`;
2. se um dos operandos é `double`, o outro é convertido para `double` e o resultado é `double`;
3. se um dos operandos é `long`, o outro é convertido para `long` e o resultado é `long`;
4. se um dos operandos é `unsigned`, o outro operando é convertido para `unsigned` e o resultado é `unsigned`;
5. senão, todos os operandos são `int` e o resultado é `int`.

Quando uma conversão imprevista ocorre, o compilador sinaliza esse evento para o programador com uma mensagem de aviso ou de erro. Porém, o programador pode indicar para o compilador que sabe que está fazendo uma conversão que o compilador não aceitaria normalmente. Para tanto, ele pode utilizar o operador de molde, que força a coerção entre dois tipos conforme indicado. No padrão da linguagem C, o operador de molde é aplicado com o nome do tipo entre parênteses antes da variável a ser convertida:

```
int main() {
        int a = 0xFF0;
        int* b;

        b = (int*) a;
}
```

Nesse caso, nenhuma mensagem é gerada pelo compilador. Em C++, o operador de molde utilizado no padrão C pode ser utilizado, mas há outras formas do operador que permitem um controle mais refinado daquilo que é alterado em relação ao comportamento padrão. Os operadores são `static_cast`, usado para

realizar conversões equivalentes ao operador tradicional da linguagem C, sem verificação em tempo de execução; `dynamic_cast`, usado para conversões com verificação de validade em tempo de execução; `reinterpret_cast`, para conversões de um tipo de dados para outro usualmente incompatível; e `const_cast`, para converter uma variável definida como constante (`const`) para outra sem essa restrição. No caso do exemplo anterior, a conversão que foi forçada com o operador de molde no padrão C++ seria realizada com o operador `reinterpret_cast`:

```
int main() {
        int a = 0xFF0;
        int* b;

        b = reinterpret_cast<int *> (a);
}
```

Algumas linguagens de programação permitem, como C++, definir comportamentos diferenciados a operadores segundo o tipo de argumento que recebem. Por exemplo, na expressão

```
  c << x;
```

o operador << será interpretado como o comando de deslocamento de bits à esquerda se `c` e `x` forem inteiros, mas será uma operação de saída se `c` for uma referência para um arquivo. Esse mecanismo de adequar o comportamento do operador segundo o tipo de seus operandos é denominado sobrecarga de operadores. Em geral, essas linguagens permitem também aplicar o mesmo tipo de mecanismo a rotinas. Por meio da sobrecarga de funções, o compilador seleciona entre rotinas que têm o mesmo nome aquela cuja quantidade e lista de tipos estão adequadas à forma de invocação. A diferenciação é tratada por meio do mecanismo de decoração de nomes, apresentado na seção anterior.

A verificação de fluxo de controle é outro tipo de atividade realizada durante a análise semântica. Nesse caso, o objetivo é detectar erros nas estruturas de controle de fluxo de execução, como em repetições (`for`, `do`, `while`) ou em alternativas (`if else`, `switch case`).

Para um exemplo do tipo de situação detectada na verificação de fluxo de controle, considere o seguinte fragmento de código:

```
void f2(int j, int k) {
  if (j == k)
```

```
    break;
  else
    continue;
}
```

Não há nesse código erros sintáticos ou erros associados aos usos das variáveis. No entanto, o compilador gera as seguintes mensagens de erro:

```
In function 'f2':
...: break statement not within loop or switch
...: continue statement not within a loop
```

ou seja, ele reconhece que o comando `break` só pode ser usado para quebrar a seqüência de um comando de iteração (*within loop*) ou para indicar o fim de um bloco associado à execução de um `case` (*within switch*). Da mesma forma, um comando `continue` só pode ser usado em um comando de iteração, para indicar que a iteração corrente já está encerrada e que a execução deve prosseguir com a reavaliação da condição de repetição.

A verificação de unicidade detecta situações tais como duplicação em declarações de variáveis, de componentes de estruturas e em rótulos do programa. Por exemplo, a compilação do seguinte código

```
void f3(int k) {
  struct {
    int a;
    float a;
  } x;
  float x;
  switch (k) {
  case 0x31: x.a = k;
  case '1': x = x.a;
  }
}
```

causaria a geração das seguintes mensagens de erro pelo compilador `gcc`:

```
In function 'f3':
...: duplicate member 'a'
...: previous declaration of 'x'
...: duplicate case value
```

A primeira mensagem detecta que dois membros de uma mesma definição de estrutura recebem o mesmo nome, `a`, o que não é permitido. A segunda mensagem

refere-se à situação de que há duas variáveis de mesmo nome, `x`. A terceira mensagem indica que dois `cases` em uma expressão `switch` receberam o mesmo rótulo, o que também não é permitido. Observe que, embora a forma de expressar o valor nas diferentes opções do comando `case` tenha sido diferente, o compilador verificou que `0x31` e `'1'` são duas representações distintas para um mesmo valor e acusou a situação de erro.

5.3 Exercícios

5.1 Um programador C++ codificou o seguinte módulo como parte de sua aplicação:

```
int a;
int umaFuncao() { return a; }
void umaFuncao(int v) { a = v; }
int umaFuncao(int v, int b) { a = v + b; return a; }
```

Após compilar este código com o compilador `g++`, ele analisou o código objeto associado com o aplicativo `nm` do sistema Unix, que apresenta a tabela de símbolos do módulo objeto, e encontrou as seguintes definições para os nomes das funções:

```
0000000a T _Z9umaFuncaoi
00000018 T _Z9umaFuncaoii
00000000 T _Z9umaFuncaov
```

(a) Explique, a partir desse exemplo, como esse compilador faz a decoração de nomes de funções.

(b) Antes de chegar ao programa anterior, o programador havia tentado compilar, sem sucesso, a seguinte versão de seu código, na qual também utilizava a sobrecarga de nomes de funções:

```
int a;
int umaFuncao() { return a; }
void umaFuncao(int v) { a = v; }
        int umaFuncao(int v) { a = v; return a; }
```

Explique qual foi a causa do problema que o programador encontrou nessa compilação.

5.2 Um programa em C++ contém as seguintes declarações de variáveis:

```
char c = '0';
int i = 1;
float f = 2.0;
```

Quais dos seguintes comandos causariam erros detectados na etapa de análise semântica e, nesse caso, qual tipo de verificação captura o erro?

(a) `c = i;`

(b) `i = c;`

(c) `c % i;`

(d) `c « i;`

(e) `f « i;`

(f) `f % c;`

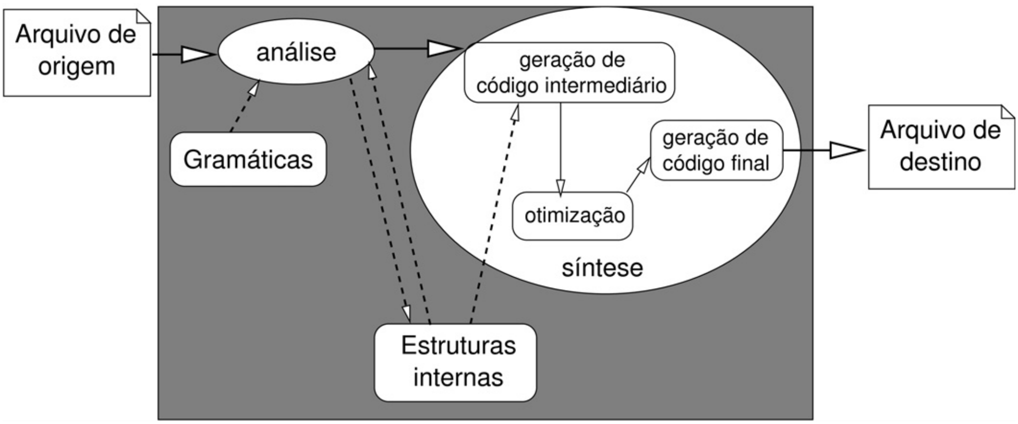
Capítulo 6

Geração de código

Os últimos capítulos apresentaram técnicas com o embasamento teórico e conceitual para permitir reconhecer os símbolos e as expressões tipicamente utilizadas em linguagens de programação de alto nível. No entanto, a operação de um compilador requer mais que o simples reconhecimento da validade de um programa; é preciso gerar o código equivalente que será efetivamente executado pelos processadores. A Figura 6.1 detalha as atividades relativas à etapa de síntese dentre as atividades realizadas pelo compilador.

A produção de código a partir da representação interna da estrutura do programa, produzida na etapa de análise sintática, tem por objetivo básico a criação

Figura 6.1 Atividades de síntese na compilação



do arquivo com o código objeto que, na etapa posterior de ligação, será combinado com outros módulos do mesmo tipo para a construção de um módulo executável. Há, porém, situações nas quais a linguagem-alvo da produção de código não é código binário, mas uma outra linguagem de alto nível que é usada como linguagem intermediária para a geração do código.

Mesmo nas situações nas quais o objetivo da geração de código é a produção do código binário, em geral esse processo utiliza etapas intermediárias que dividem a tarefa complexa em problemas menores e mais facilmente tratáveis. Em geral, a produção do código objeto é delegada ao programa montador, que traduz um arquivo em formato texto contendo o programa em linguagem simbólica para o código de máquina.

A produção do arquivo com o código em linguagem simbólica deve levar em conta as características do processador-alvo, tais como qual é o jogo de instruções disponíveis e a arquitetura do processador, como registradores e modos de endereçamento disponíveis. Essa produção de código é facilitada se houver uma descrição intermediária do programa, não tão abstrata como a árvore sintática nem tão específica como a linguagem simbólica. Essa tarefa de geração de código em formato intermediário é descrita na Seção 6.1.

Como a produção automática de código em formato intermediário é realizada a partir da tradução de fragmentos associados aos comandos básicos e tratados de forma isolada, é usual que haja no código assim produzido elementos desnecessários ou redundantes. Uma etapa de otimização de código, descrita na Seção 6.2, realiza a análise do código criado com a combinação desses fragmentos e produz um código com semântica equivalente e que favorece algum critério de otimização, como por exemplo um menor tempo de execução.

Finalmente, ocorre a tradução do código em formato intermediário para o código em linguagem simbólica de um processador específico. Nessa etapa, descrita na Seção 6.3, ocorre a seleção de instruções fornecidas pelo processador. Em alguns casos, o código traduzido nessa última etapa de compilação é ainda passível de otimização, nas situações em que o processador permite mais de uma alternativa para realizar uma mesma tarefa.

6.1 Geração de código intermediário

A tradução do código de alto nível para o código do processador está associada a traduzir para a linguagem-alvo dos comandos elementares encontrados no código. Durante a análise sintática, esses comandos elementares já foram re-

conhecidos — por exemplo, na construção dos nós internos da árvore sintática obtida para as diversas expressões do programa. Assim, é possível associar a análise sintática à tarefa de construir a árvore sintática e depois percorrer essa representação para produzir código na linguagem simbólica de destino. Embora essa produção de código possa ocorrer assim, em geral ela é realizada durante a execução das ações semânticas associadas à aplicação das regras de reconhecimento do analisador sintático. Esse procedimento é denominado tradução dirigida pela sintaxe.

Em geral, a geração de código não se dá diretamente para a linguagem simbólica do processador-alvo. Por conveniência, o analisador sintático gera código para uma máquina abstrata, com uma linguagem com comandos similares àqueles de uma linguagem simbólica porém independente de processadores específicos. Em uma segunda etapa de geração de código, esse código em linguagem intermediária é traduzido para um código equivalente na linguagem simbólica desejada. Dessa forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores, pois todos os módulos anteriores à produção de código final são independentes dessa etapa.

A linguagem utilizada para a geração de um código em formato intermediário entre a linguagem de alto nível e a linguagem simbólica deve representar, de forma independente do processador para o qual o programa será gerado, todas as expressões do programa original. Duas formas usuais para esse tipo de representação são a notação pós-fixa e o código de três endereços.

6.1.1 Código de três endereços

O linguagem intermediária em código de três endereços define seqüências de instruções envolvendo operações com uma atribuição ou instruções de desvio. O nome "três endereços" está associado à especificação, em uma instrução que representa uma operação binária, de no máximo três variáveis: duas para os operadores e uma para o resultado. Assim, expressões complexas envolvendo diversas operações devem ser decompostas nessa linguagem em uma série de instruções elementares. Eventualmente, pode ser necessário utilizar variáveis temporárias que devem ser introduzidas nesse processo de tradução para manter os valores intermediários das subexpressões.

O código nessa linguagem intermediária tem uma estrutura próxima daquela dos programas em linguagem simbólica, cujas instruções representam as operações elementares do processador. Por esse motivo, um programa em linguagem intermediária pode ser facilmente convertido para uma linguagem-alvo.

Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções: expressões com atribuição, desvios, invocação de rotinas e acesso indexado e indireto.

Instruções de atribuição

As instruções de atribuição são aquelas nas quais o resultado de uma operação é armazenado na variável especificada à esquerda do operador de atribuição, aqui denotado por $:=$. Apesar da diversidade de formatos que esse operador pode assumir em linguagens de alto nível, como em C ou C++, há três formas básicas para esse tipo de instrução que são utilizadas no formato intermediário de código.

A forma básica, mais simples, é aquela que realiza a cópia de valor de uma variável para outra:

```
le := ld
```

Outra forma usual do operador ocorre quando ele é utilizado para armazenar o resultado de uma operação binária:

```
le := ld1 op ld2
```

O resultado a ser atribuído ao lado esquerdo da instrução de atribuição pode ser também obtido a partir da aplicação de um operador unário a uma outra variável:

```
le := op ld
```

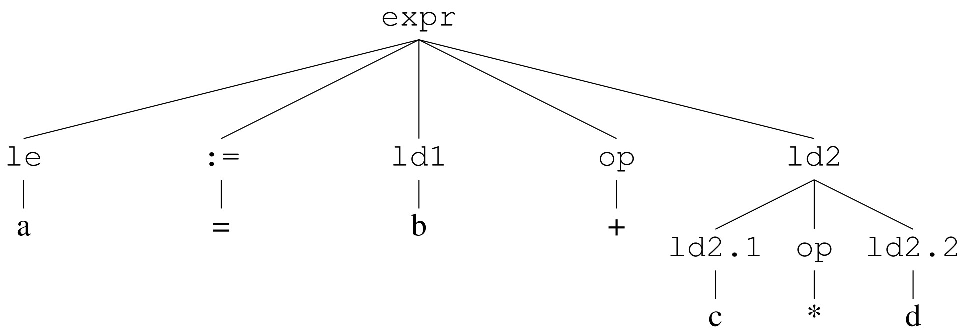
Algumas expressões de atribuição em C++ seriam traduzidas de forma imediata para o correspondente nesse formato intermediário, como

```
a = c;
b = a + d;
e = -b;
```

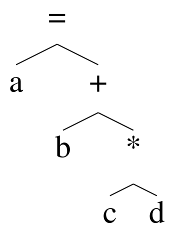
Expressões mais complexas demandam a utilização de variáveis intermediárias para permitir a expressão no formato intermediário. Por exemplo, considere a expressão

```
a = b + c * d;
```

A árvore sintática para essa expressão pode ser representada como



Para a geração de código, uma estrutura de árvore simplificada — a árvore sintática abstrata — é adotada. Nessa representação, os operadores são representados nos nós internos:



É possível observar que o nó rotulado * é utilizado como um dos operandos em outra operação binária. Na geração do código intermediário, esse nó deve ser associado a uma variável temporária — por exemplo, `_t1`. Com essa associação, a expressão completa seria traduzida para o formato intermediário por duas instruções:

```
_t1 := c * d
  a := b + _t1
```

Instruções de desvio

As instruções de desvio podem assumir duas formas básicas. Uma instrução de desvio incondicional tem o formato

```
goto L
```

onde `L` é um rótulo que identifica uma linha do código.

A outra forma de desvio é o desvio condicional, com o formato

```
if x opr y goto L
```

onde `opr` é um operador relacional de comparação e `L` é o rótulo da linha que deve ser executada se o resultado da aplicação do operador relacional for verdadeiro; caso contrário, a linha seguinte é executada.

Por exemplo, a seguinte iteração em C++

```
while (i++ <= k)
    x[i] = 0;
x[0] = 0;
```

pode ser traduzida no formato de código intermediário para a seguinte seqüência de instruções:

```
_L1: if i > k goto _L2
     i := i + 1
     x[i] := 0
     goto _L1
_L2: x[0] := 0
```

Instruções de invocação de sub-rotinas

A invocação de rotinas, na programação em linguagem simbólica, é normalmente realizada por uma instrução que altera o ponto de execução para outra região da memória ao mesmo tempo que preserva o ponto de execução atual para restaurá-lo quando a sub-rotina é concluída. A passagem de argumentos e valores de retorno é tratada explicitamente pelo programador, que deve manipular o conteúdo da pilha para esse fim.

Na linguagem de código intermediário, a instrução `call` realiza a invocação da rotina. Seu primeiro argumento é o nome da rotina; na linguagem simbólica, essa é a informação necessária para codificar a instrução de invocação. A restauração do ponto de execução após a conclusão da sub-rotina é realizada por uma instrução `return`, que também tem mapeamento direto para instruções equivalentes em linguagem simbólica.

O ponto no qual a linguagem intermediária em código de três endereços é um pouco mais simples que a tarefa equivalente realizada com a programação em linguagem simbólica é na passagem de argumentos e de valores de retorno. Na linguagem de alto nível, a invocação tem um valor de retorno que pode ser usado

numa atribuição e os argumentos de uma função são simplesmente listados entre parênteses, como no seguinte fragmento de código C++:

```
x = f(a, b, c);
```

Já na programação em linguagem simbólica, argumentos e resultados de uma sub-rotina são transferidos por meio da pilha, a área de memória que mantém, durante a execução de um programa, valores temporários associados à seqüência de rotinas invocadas. Para abstrair os detalhes de operação da pilha sem ficar muito distante desse mecanismo de passagem de informação, a linguagem de código intermediário introduz a instrução `param`, que prepara cada um dos argumentos para uma invocação que será executada na seqüência. Por exemplo, na invocação da rotina do exemplo citado, como a rotina tem três argumentos, haveria três linhas com a instrução `param`, na mesma seqüência da lista de argumentos:

```
param a
param b
param c
```

Na instrução de invocação na linguagem intermediária, além do nome da rotina, a instrução `call` tem um segundo parâmetro que é a quantidade de elementos registrados com `param` que serão consumidos pela rotina:

```
...  call f, 3
```

Resta apenas definir como o valor de retorno é obtido na programação do código em linguagem intermediária. Nas linguagens nas quais as rotinas podem retornar apenas um valor, o formato mais simples é simplesmente ter esse resultado associado à instrução `call` e assim permitir a atribuição direta desse valor de retorno a uma variável. Desse modo, a tradução completa da instrução de invocação da rotina de alto nível para o formato de código em linguagem intermediária é:

```
param a
param b
param c
x := call f,3
```

Considere um outro exemplo no qual, na linguagem de alto nível, um dos argumentos de uma função é uma outra função, como em

```
a = g (b, h(c));
```

Na linguagem de código intermediário, essa instrução será codificada como

```
param b
param c
_t1 := call h, 1
param _t1
a := call g, 2
```

Nesse caso, a rotina `h` consome um único argumento, que foi o último elemento passado numa instrução `param` — nesse caso, `c`. Já a rotina `g` consome os dois elementos que foram registrados com `param` e que ainda não foram consumidos por outras rotinas.

Modos de endereçamento

Os exemplos de instruções no formato de código intermediário de três endereços até o momento utilizaram o modo de endereçamento direto. Adicionalmente, os modos de endereçamento indexado e indireto podem ser utilizados. Por simplicidade, na linguagem de formato intermediário aqui definida esses dois modos estão associados a atribuições apenas, muito embora seja usual em linguagens simbólicas que esses modos sejam usados em combinação com outras instruções.

O modo de endereçamento indireto está associado à manipulação de variáveis que contêm endereços. Na linguagem de código intermediário, os operadores utilizados são os mesmos da linguagem C++. Assim, o operador `&` permite obter o endereço de uma variável. Por exemplo,

```
x := &y
```

atribui o endereço da variável `y` à variável `x`.

O operador `*` faz a operação inversa. A instrução

```
w := *x
```

atribui o valor que está armazenado no endereço `x` à variável `w`.

Também é possível atribuir um valor à variável cujo endereço é conhecido, como em

```
*x := z
```

que atribui o valor da variável z à posição endereçada por x. Observe que a definição do endereço de uma variável não é atribuição do programador, de modo que uma instrução na forma abaixo não faz sentido:

```
&y := t
```

Instruções na linguagem de alto nível que utilizam endereçamento indireto devem ser decompostas de forma a isolar, no formato intermediário, esse tipo de endereçamento apenas nas operações de atribuição, como foi definido para esse formato. Por exemplo, considere a seguinte instrução C++, na qual a é um valor escalar do tipo inteiro e p1 e p2 são duas variáveis ponteiros para o tipo inteiro:

```
*p1 = a + *p2++;
```

Nessa instrução, o conteúdo da posição indicada por p2 é somado ao valor de a; o resultado dessa operação é atribuído ao conteúdo da posição indicada por p1. Além disso, como a variável p2 está associada ao operador de pós-incremento ++, seu valor é incrementado de uma posição. Se nesse compilador o valor inteiro foi definido como sendo de quatro bytes, então o valor de p2 deve ser incrementado em quatro.

No formato intermediário, essa expressão é traduzida para:

```
_t1 := *p2
 p2 := p2 + 4
_t2 := a + _t1
*p1 := _t2
```

O endereçamento indexado é aquele no qual a posição do item de informação acessado é definida a partir da informação de um endereço-base e de um deslocamento (o índice). Assim, a instrução

```
x := y[i]
```

transfere para a variável x o conteúdo do endereço indicado pela soma da variável y com o valor i.

Ao contrário do que acontece nas linguagens de alto nível, na linguagem simbólica as referências a esses deslocamentos associados aos índices devem ser

traduzidas para bytes. Por exemplo, se em C++ um arranjo `z` for declarado como um arranjo de inteiros e se inteiros são de quatro bytes, então uma referência à variável `z[5]` leva à posição que está 20 bytes adiante da posição de `z`. No código de formato intermediário, essa informação já deve estar presente.

Da mesma forma que é possível ler o conteúdo de uma variável indexada, é possível atribuir valores a ela, como em

```
x[i] := y
```

Considere o seguinte laço em C++, com `a`, `b` e `s` sendo arranjos com dez inteiros de quatro bytes cada:

```
for (i=0; i<10; ++i)
  s[i] = a[i] + b[i];
```

A tradução desse fragmento de código para a linguagem de formato intermediário resulta em

```
      i := 0
_L1:  if i >= 10 goto _L2
      _t1 := 4*i
      _t2 := a[_t1]
      _t3 := 4*i
      _t4 := b[_t3]
      _t5 := _t2 + _t4;
      _t6 := 4*i
      s[_t6] := _t5
      i := i + 1
      goto _L1
_L2:  ...
```

Como é possível observar nesse fragmento, há muitas redundâncias que poderiam ser facilmente eliminadas com uma análise do código gerado. Tal aspecto da produção de código será abordado na Seção 6.2.

Representação interna

Embora o código em formato intermediário possa, em princípio, ser armazenado em um arquivo em formato texto, o processamento desse arquivo exigiria

novamente a realização de tarefas associadas ao compilador. Assim, a representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em estruturas de tabelas.

Uma abordagem de armazenamento das instruções em formato intermediário utiliza quádruplas, que são tabelas com quatro colunas. Nesse caso, cada instrução é representada por uma linha na tabela com a especificação de quatro elementos:

1. o operador da linguagem de formato intermediário;
2. o primeiro argumento;
3. o segundo argumento, se presente;
4. o resultado, se presente.

Por exemplo, considere a tradução das expressões C++:

```
a = b + c * d;
z = f (a);
```

No formato de código intermediário, as duas expressões são traduzidas para

```
_t1 := c * d
a := b + _t1
param a
z := call f, 1
```

Armazenado no formato de quádruplas, esse fragmento teria a seguinte representação na tabela:

	operador	arg 1	arg 2	resultado
1	*	c	d	_t1
2	+	b	_t1	a
3	param	a		
4	call	f	1	z

Como é possível observar, há instruções para as quais algumas das colunas da tabela ficam vazias.

Instruções de desvio podem não ter uma definição, no momento da tradução, da posição associada ao destino. Nesses casos, é usual utilizar a técnica de retrocorreção. A estratégia de definição de rótulos de desvio por retrocorreção cria,

durante a tradução, uma lista auxiliar com as pendências de destino na tabela. À medida que esses destinos são definidos, a informação é inserida na lista. Ao final da tradução, a lista é percorrida para preencher as lacunas na tabela.

Além da representação por quádruplas, instruções na linguagem de código intermediário também podem ser armazenadas na forma de triplas. Nesse caso, uma tabela de três colunas é utilizada. A informação que é omitida é a variável com o resultado, que em geral faz referências a variáveis temporárias. Na representação por triplas, os resultados de expressões são indicados por meio de referências às suas posições na tabela.

Considere novamente o exemplo do fragmento de código acima. O mesmo código em linguagem de formato intermediário pode ser representado na forma de triplas com

	operador	arg 1	arg 2
1	`*`	`c`	`d`
2	`+`	`b`	(1)
3	`:=`	`a`	(2)
4	`param`	`a`	
5	`call`	`f`	`1`
6	`:=`	`z`	(5)

Observe que, na representação por triplas, a atribuição a uma variável não-temporária deve ser representada explicitamente na tabela, enquanto na representação por quádruplas a atribuição está implícita na representação do resultado.

Notação pós-fixa

A notação tradicional para expressões aritméticas, que representa uma operação binária na forma `x+y`, ou seja, com o operador entre seus dois operandos, é conhecida como notação infixa. Uma notação alternativa para esse tipo de expressão é a notação pós-fixa, também conhecida como notação polonesa, na qual o operador é expresso após seus operandos.

O atrativo da notação pós-fixa é que ela dispensa o uso de parênteses ao adotar a noção de pilha para a representação das expressões. Assim, quando um operador binário aparece na seqüência nessa representação, ele é aplicado aos dois últimos elementos e o resultado de sua aplicação está disponível como um novo elemento. Há algumas calculadoras eletrônicas que adotam esse formato para sua operação.

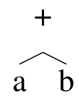
Por exemplo, a expressão que na notação infixa, tradicional, é representada como

```
a + b
```

na notação pós-fixa é representada por

```
a b +
```

Observe que ambas representam a mesma expressão, cuja árvore sintática abstrata é

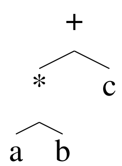


que, na notação infixa, foi percorrida com a estratégia de varredura intra-ordem e na notação pós-fixa foi percorrida com a estratégia pós-ordem.

Similarmente, considere a expressão em notação infixa

```
a * b + c
```

que corresponde à árvore sintática abstrata



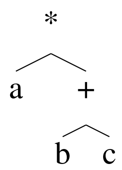
Ao percorrer essa árvore com a estratégia pós-ordem, obtém-se a notação pós-fixa

```
a b * c +
```

Considere a expressão a seguir que, na notação infixa, requer parênteses para alterar a precedência:

```
a * ( b + c )
```

A árvore sintática abstrata que representa a estrutura dessa expressão é



Percorrida em pós-ordem, resulta na notação pós-fixa, sem necessidade de parênteses:

```
a b c + *
```

Instruções de desvio em código intermediário também podem ser representadas com a notação pós-fixa. Nesse caso, assumem as formas

```
L jump
x y L jcc
```

para desvios incondicionais e condicionais, respectivamente. No caso de um desvio condicional, a condição a ser avaliada envolvendo `x` e `y` é expressa na parte `cc` da própria instrução. Assim, `jcc` pode ser uma instrução entre `jeq` (desvio ocorre se `x` e `y` forem iguais), `jne` (se diferentes), `jlt` (se `x` menor que `y`), `jle` (se `x` menor ou igual a `y`), `jgt` (se `x` maior que `y`) ou `jge` (se `x` maior ou igual a `y`).

6.2 Otimização de código

O código gerado pela tradução orientada a sintaxe contempla cada expressão do código-fonte individualmente. Ao avaliar globalmente o código assim gerado, é possível observar a existência de diversas seqüências de código contendo trechos ineficientes. O objeto da etapa de otimização de código é aplicar um conjunto de heurísticas para detectar tais seqüências e substituí-las por outras que removam essas situações de ineficiência.

Há dois momentos nos quais as técnicas de otimização podem ser aplicadas. As técnicas de otimização independentes de máquina podem ser aplicadas no código em formato intermediário, antes da produção do código em linguagem simbólica. As técnicas de otimização que dependem do conhecimento do jogo de instruções específico de um processador, para saber quais instruções podem realizar uma mesma tarefa de forma mais eficiente, são denominadas técnicas dependentes de máquina. Nesta seção serão abordadas as técnicas de otimização independentes de máquina.

As técnicas de otimização aplicadas pelo compilador utilizam análises dos blocos de instruções do programa gerado. Nessas análises, o programa é representado por um grafo cujos nós são os blocos de instruções e os arcos são os possíveis caminhos de execução. Dentro de cada bloco, a definição e o uso das variáveis são analisados para descobrir possibilidades de utilização de técnicas

de otimização associadas aos seus valores. A etapa da otimização que realiza esses levantamentos é conhecida como análise de fluxo, que por sua vez contempla dois aspectos, a análise de fluxo de controle e a análise de fluxo de dados. Estratégias que podem ser aplicadas ao analisar um único bloco de comandos são denominadas estratégias de otimização local, enquanto aquelas que envolvem a análise simultânea de dois ou mais blocos são denominadas estratégias de otimização global.

Considere novamente o trecho de código gerado no corpo do laço com acesso indexado, apresentado na seção anterior:

```
 _t1 := 4*i
 _t2 := a[_t1]
 _t3 := 4*i
 _t4 := b[_t3]
 _t5 := _t2 + _t4;
 _t6 := 4*i
s[_t6] := _t5
```

Como o valor de `i` não é alterado ao longo da seqüência de instruções desse bloco, as variáveis `_t1`, `_t3` e `_t6` têm sempre o mesmo valor. Portanto, ao verificar que `_t3` e `_t6` podem ser substituídas por `_t1`, o módulo de otimização pode substituir essa seqüência por

```
 _t1 := 4*i
 _t2 := a[_t1]
 _t4 := b[_t1]
 _t5 := _t2 + _t4;
s[_t1] := _t5
```

Esse é um exemplo de uma das possíveis técnicas de otimização, a eliminação de subexpressões comuns. No código gerado automaticamente, várias dessas possibilidades estão presentes.

Outra heurística aplicada à otimização é a estratégia de eliminação de código redundante. O objetivo dessa estratégia é detectar situações nas quais a tradução de duas expressões gera instruções que, combinadas, têm uma execução repetida sem efeito. Por exemplo, considere a situação na qual as duas expressões a seguir foram geradas:

```
le := ld
   ...
le := ld
```

Aparentemente, a segunda instrução é redundante e pode ser eliminada. A análise de fluxo de dados irá determinar se este é mesmo o caso ao avaliar o que ocorre entre as duas instruções. Por exemplo, se o valor de `ld` foi alterado entre as duas instruções, então o segundo `ld` é diferente do primeiro e a instrução não é redundante e, portanto, deve ser mantida. A análise de fluxo de dados representaria esse fragmento como

```
le := ld1
...
ld2 := x
...
le := ld2
```

Se o valor de `ld` não se altera mas o de `le` sim, então ainda não há uma definição. Por exemplo, se houver algum uso da variável `le` alterada, a última atribuição é uma restauração do valor original e deve ser preservada. Caso contrário, a atribuição intermediária é desnecessária e pode ser eliminada, assim como a atribuição redundante.

O mesmo princípio de eliminação seria aplicável se o bloco de instruções fosse

```
le := ld
...
ld := le
```

e o valor de `le` não fosse modificado entre as duas instruções.

Outra técnica de análise de fluxo de dados que explora a igualdade de valores entre variáveis distintas é a propagação de cópias. Considere a ocorrência em um bloco de um padrão na forma

```
le1 := ld
...
le2 := le1
```

Se a variável `le1` não está viva na saída desse bloco, ou seja, não há outros usos desse valor que lhe foi atribuído no bloco, e se não há alterações no valor de `ld` entre essas duas atribuições, a variável `le1` pode ser eliminada e seu uso na segunda atribuição pode ser substituído pelo uso da variável `ld`:

```
...
le2 := ld
```

Outras heurísticas de otimização estão relacionadas à análise do fluxo de controle do programa. Por exemplo, considere a tradução de duas expressões consecutivas no programa original na qual o final da primeira expressão seja um desvio incondicional para a primeira linha da instrução seguinte. Ao concatenar o código assim gerado, o resultado é algo da forma

```
     <a>
     goto _L1
_L1: <b>
```

Nesse caso, a linha com a instrução `goto` é a última do código associado ao bloco de instruções `<a>` e a linha com o rótulo `_L1` é a primeira do bloco de instruções `<b>`. Esse código pode ser seguramente reduzido com a aplicação da técnica de eliminação de desvios desnecessários — como a instrução após `goto` seria executada independentemente do desvio ocorrer, ela é desnecessária. O resultado da eliminação dessa linha gera a seqüência equivalente

```
     <a>
_L1: <b>
```

Outra estratégia de otimização elimina os rótulos não referenciados por outras instruções do programa. Assim, se o rótulo `_L1` estivesse sendo referenciado exclusivamente por essa instrução de desvio, ele poderia ser eliminado em uma próxima aplicação das estratégias de otimização.

Com a análise do fluxo de controle também é possível aplicar a estratégia de eliminação de código não-alcançável ou "código morto". Por exemplo, considere a seguinte seqüência de código em linguagem de formato intermediário gerada na tradução de um conjunto de expressões da linguagem de alto nível:

```
       ...
     goto _L1
     <a>
_L1:   ...
```

Caso nenhuma linha do bloco de instruções `<a>` seja alvo de uma instrução de desvio, não há como as instruções desse bloco serem executadas. Sem essa possibilidade de ter um ponto de entrada para a sua execução, todas as instruções desse bloco podem ser eliminadas, pois pelo fluxo seqüencial de execução o bloco nunca será alcançado.

O uso de propriedades algébricas é outra estratégia de otimização usualmente aplicada. Uma primeira situação é quando o módulo otimizador detecta

que uma expressão binária tem, como um dos operandos, o elemento identidade daquela operação. Nesse caso, ele pode eliminar a expressão e substituí-la simplesmente pelo outro operando, como em

Substituir	Por
x + 0	x
0 + x	x
x - 0	x
x * 1	x
1 * x	x
x / 1	x

Outras propriedades algébricas são utilizadas com o objetivo de substituir operações de alto custo de execução por operações mais simples. Por exemplo, como o cálculo da potência é mais complexo que a execução de uma multiplicação, o módulo otimizador pode substituir, em uma expressão para computar o valor de x^2, a invocação da função `pow(x,2)` da linguagem C++ pela expressão `x*x`, que produz o mesmo resultado.

Do mesmo modo, é possível substituir, no caso de variáveis reais, `2*x` por `x+x` ou substituir `x/2` por `0.5*x`. Se `x` for uma variável inteira, a divisão por dois pode ser substituída por um deslocamento da representação binária à direita de um bit. Genericamente, a divisão inteira por 2^n equivale ao deslocamento à direita de n bits na representação binária do dividendo. Da mesma forma, a multiplicação inteira por potências de dois pode ser substituída por deslocamento de bits à esquerda.

No caso de operações comutativas e associativas, o uso de suas propriedades algébricas permite também o reuso de subexpressões já computadas. Por exemplo, a tradução direta das expressões

```
a = b + c;
e = c + d + b;
```

geraria o seguinte código intermediário:

```
  a := b + c
_t1 := c + d
  e := _t1 + b
```

No entanto, o uso dessas propriedades para a adição permite que o código gerado seja reduzido usando a eliminação de expressões comuns, resultando em

```
a := b + c
e := a + d
```

Diversas oportunidades de otimização estão associadas à análise de comandos iterativos. Uma estratégia é a movimentação de código, aplicado quando um cálculo realizado dentro do laço envolve valores invariantes na iteração. Por exemplo, considere o seguinte comando de iteração em C++:

```
while (i < 10*j) {
  a[i] = i + 2*j;
  ++i;
}
```

Se o arranjo a é de inteiros de quatro bytes, o seguinte código em formato intermediário seria gerado, sem otimização:

```
_L1:  _t1 := 10 * j
      if i >= _t1 goto _L2
      _t2 := 2 * j
      _t3 := i + _t2
      _t4 := 4 * i
      a[_t4] := _t3
      i := i + 1
      goto _L1
_L2:    ...
```

No entanto, a análise de definição e uso da variável j identifica que seu valor não é alterado dentro do laço e, portanto, a variável _t2 é constante entre iterações. O compilador pode mover as expressões que envolvem exclusivamente constantes na iteração para fora do laço e, nesse caso, substituir esse código por

```
      _t1 := 10 * j
      _t2 := 2 * j
_L1:  if i >= _t1 goto _L2
      _t3 := i + _t2
      _t4 := 4 * i
      a[_t4] := _t3
      i := i + 1
      goto _L1
_L2:    ...
```

Nesse caso, todas as instruções no bloco interno ao laço têm dependência em relação ao valor da variável de iteração `i`.

As técnicas de otimização que são usadas em compiladores devem, além de manter o significado do programa original, ser capazes de capturar a maior parte das possibilidades de melhoria do código dentro de limites razoáveis de esforço gasto para tal fim. Em geral, os compiladores usualmente permitem especificar qual o grau de esforço desejado no processo de otimização. Por exemplo, o compilador `g++` permite na linha de comando o uso de uma opção `-O`, de otimização, que dá essa indicação. A omissão dessa opção ou a utilização da opção `-O0` (a letra O seguida do dígito zero) indica ao compilador que nenhuma otimização deve ser feita. As opções `-O1` e `-O2` introduzem gradualmente no processo de compilação algumas heurísticas de otimização; a opção `-O3` indica ao compilador para aplicar ao máximo as heurísticas de otimização e, conseqüentemente, demanda um maior tempo de compilação. Tais otimizações referem-se à minimização do tempo de execução. Para algumas aplicações, como no caso de programas que são desenvolvidos para dispositivos portáteis, o critério de otimização mais interessante pode ser a redução do espaço de memória ocupado e não tanto o tempo de execução. Para esse caso, há a opção `-Os` cujo objetivo é reduzir a ocupação de espaço em memória. Qualquer que seja a opção, a adoção de otimização pode dificultar o trabalho de depuração.

6.3 Geração de código em linguagem simbólica

A última etapa do compilador propriamente dito é a geração do código em linguagem simbólica. Uma vez que esse código seja gerado, outro programa — o montador — será responsável pela tradução para o código objeto, em formato de linguagem de máquina. O formato adotado para o código intermediário, ao limitar o número de operadores e os tipos de instruções, tem por objetivo exatamente facilitar a produção de código em linguagem simbólica.

A abordagem mais simples para essa etapa é ter, para cada instrução do formato intermediário, um gabarito com a correspondente seqüência de instruções em linguagem simbólica do processador-alvo. Por exemplo, considere a instrução em formato intermediário:

```
le := ld1 + ld2
```

A seqüência de instruções em linguagem simbólica que corresponde a essa instrução depende da arquitetura do processador para o qual o programa é gerado. Para esse processo, uma das características importantes a considerar do

processador é o número de operandos com que a instrução opera. Em relação a esse aspecto, há uma classificação de processadores de acordo com o número de operandos especificados em uma operação binária na linguagem simbólica.

Máquinas de três endereços são aquelas nas quais as instruções correspondentes a operações binárias explicitam os endereços dos dois operandos de entrada, assim como o endereço no qual o resultado será armazenado. Por exemplo, uma operação de soma `z=a+b` numa máquina desse tipo poderia ser codificada tipicamente por uma instrução na forma

```
ADD  a, b, z
```

ou

```
ADD z, a, b
```

Numa máquina de dois endereços, as instruções correspondentes a operações binárias explicitam apenas os endereços dos dois operandos de entrada. O resultado é implicitamente assumido como sendo o mesmo do primeiro operando. Por exemplo, uma instrução típica nesse tipo de máquina seria

```
ADD a, b
```

que produz o resultado `a = a + b`. Tipicamente, nesse caso, o primeiro operando não é uma variável do programa, mas sim um registrador do processador; desse modo, o valor original da variável não é alterado. Uma instrução de transferência entre memória e registrador, como `MOVE`, é normalmente utilizada para preparar o valor desse registrador para a operação e para salvá-lo na memória, quando necessário. Assim, uma operação `z = a + b` seria traduzida por uma seqüência de instruções como

```
MOVE a, R0
ADD  R0, b
MOVE R0, z
```

Nesse caso, foi assumido que o primeiro argumento da instrução `MOVE` é a origem da transferência e o segundo argumento, seu destino. Dependendo do processador, a forma de codificação pode ser ao contrário.

Uma máquina de um endereço é aquela cujas instruções correspondentes a operações binárias especificam apenas o endereço de um operando, usualmente o segundo. O endereço do primeiro operando e o endereço do resultado são implicitamente assumidos como sendo um registrador especial, o acumulador. Tipicamente, nesse tipo de processador a transferência da memória para o

acumulador é realizada por uma instrução `LOAD`, enquanto a transferência do acumulador para a memória é feita por uma instrução `STORE`. A seqüência de instruções para somar `a` e `b` e armazenar o resultado em `z` é, nesse tipo de máquina, tipicamente da forma:

```
LOAD   a
ADD    b
STORE  z
```

Há também processadores que utilizam uma arquitetura de zero endereços. Nesse caso, a instrução para realizar a operação binária não explicita nenhum endereço. O processador assume que os operandos são retirados de uma pilha, que pode ser de registradores ou na memória, com o resultado também armazenado na pilha. Tipicamente, nesse tipo de arquitetura, os operandos são transferidos da memória para a pilha com uma operação `PUSH` e da pilha para a memória com uma instrução `POP`. Obviamente, essas duas instruções especificam um endereço. Um fragmento de código típico para executar a operação de soma usada como exemplo nos casos anteriores é:

```
PUSH a
PUSH b
ADD
POP  z
```

A tradução de código do formato intermediário para o formato de linguagem simbólica nesse tipo de máquina pode ser simplificada pela adoção do formato intermediário usando a notação pós-fixa.

Uma vez estabelecida a arquitetura da máquina, o compilador precisa apenas manter os gabaritos para produzir o código em linguagem simbólica associado às instruções básicas da linguagem de formato intermediário. Como as instruções nesse formato são simples, já limitadas a dois operandos, fazer tal tradução é simples.

No entanto, o código gerado por esse processo automático de tradução pode também causar redundâncias, que podem ser eliminadas numa etapa de otimização do código final, já em linguagem simbólica. Por exemplo, considere a tradução das instruções

```
a := b + c
d := a + e
```

para uma máquina de dois endereços:

```
MOVE b, RO
ADD  R0, c
MOVE R0, a
MOVE a, R0
ADD  R0, e
MOVE R0, d
```

Como pode ser observado, a quarta instrução dessa seqüência é desnecessária, pois seu efeito é redundante com a instrução anterior, e portanto pode ser simplesmente removida do código gerado.

A seleção de instruções nessa fase também dá margem a outra forma de otimização, quando o processador oferece mais de uma alternativa para realizar a mesma operação. São as otimizações dependentes de máquina. Por exemplo, considere a instrução

```
x := y + K
```

na qual `K` é alguma constante inteira. A tradução para o código em linguagem simbólica de um processador da família 68K, da Motorola, assume genericamente a forma

```
MOVE.L    y,D0
ADDI.L    #K,D0
MOVE.L    D0,x
```

No entanto, se o compilador verificasse que o valor de `K` era uma constante entre 1 e 8, então numa estratégia de otimização para redução de espaço a segunda instrução poderia ser substituída por `ADDQ.L`, cuja codificação requer apenas dois bytes em vez dos seis bytes necessários para codificar a instrução `ADDI.L`.

6.4 Exercícios

6.1 Para cada uma das expressões C a seguir, para as quais todas as variáveis são do tipo inteiro de quatro bytes:

```
x = ( a + b + c ) * ( b + c ) - d;
r = a * b + pow(c,2);
b[i] = a[i] + a[i-1]
```

(a) Escreva o bloco de instruções em código intermediário de três endereços.

(b) Para cada bloco, escreva um bloco equivalente otimizado. Indique claramente quais técnicas de otimização foram utilizadas.

(c) Apresente a representação do código otimizado em notação de quádruplas para cada expressão.

(d) Repita o item anterior usando a notação de triplas.

6.2 Para o código otimizado gerado para cada expressão na questão anterior, apresente o código em linguagem simbólica gerado para um processador com instruções aritméticas `ADD`, `SUB` e `MUL` e com arquitetura de

(a) três endereços;

(b) dois endereços;

(c) um endereço;

(d) zero endereços.

6.3 Para cada uma das expressões aritméticas a seguir, apresente a representação equivalente em notação pós-fixa:

(a) $(a + b + c) \times (b + c) - d$

(b) $a - (b - a \times (c + b/d))$

(c) $(a + b) - (a - (c - d) \times (e - f) + g)/h$

6.4 Apresente o código em formato intermediário de três endereços para o seguinte fragmento de código C:

```
i = 0;    /* decl: int i */
while (i < 10) {
   a[i] = 0;  /* decl: int a[10] */
   i = i + 1;
}
```

6.5 Para o seguinte fragmento de código C, com as variáveis do tipo `int` (quatro bytes):

```
for (i = 0; i < 100; ++i) {
   eps = 0.001;
   if (a[i] != 0)
      c[i] = b[i] / a[i];
   else
      c[i] = eps;
}
```

(a) Apresente o código em formato intermediário de três endereços, sem otimização.

(b) Otimize o código em formato intermediário, indicando claramente quais foram as estratégias utilizadas.

Capítulo 7

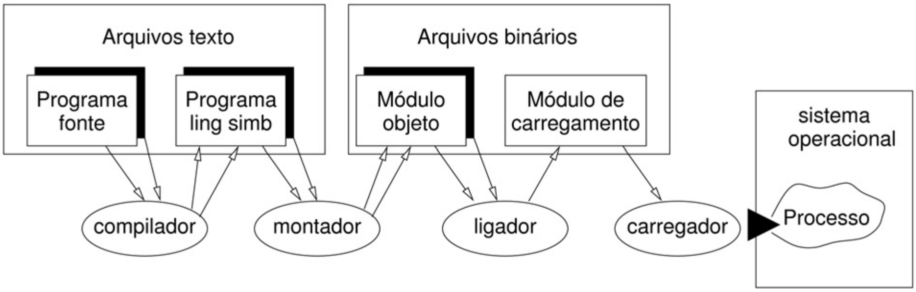
Produção do código executável

O resultado do processo de compilação é um arquivo contendo um programa em linguagem simbólica equivalente ao programa originalmente escrito em linguagem de alto nível. Esse programa em linguagem simbólica contém seqüências de instruções mnemônicas que representam as operações que são específicas de um processador. No entanto, para que esse programa possa ser executado há ainda algumas transformações adicionais pelas quais deve passar.

A primeira delas é a montagem, processo pelo qual o programa em linguagem simbólica é transformado em formato binário, em código de máquina. O programa responsável por essa transformação é o montador. Entretanto, esse processamento ainda não transforma o programa em código executável. Em geral, o programa do usuário utiliza recursos fornecidos por outras rotinas, desenvolvidos por outros — por exemplo, as rotinas para apresentar valores formatados na tela desenvolvidas pelos projetistas da linguagem de alto nível. Assim, é preciso realizar um outro processo que combina essas rotinas adicionais com o código compilado — é o processo de ligação. Finalmente, é preciso que o código seja transferido do disco para a memória principal para que sua execução possa iniciar. O módulo responsável por essa tarefa é o carregador. A Figura 7.1 resume a seqüência de transformações pelas quais passa o programa de sua concepção à sua execução.

A execução de cada programa se dá sob o controle do sistema operacional; cada programa em execução é denominado um processo do sistema. Além do código gerado pelo compilador, um processo necessita de informações adicionais para o controle de sua execução. O conjunto dos valores dessas informações associadas a cada programa em execução constitui o estado do processo. O sis-

Figura 7.1 Etapas para a criação e a execução de programa



tema operacional é o responsável por gerenciar cada processo no computador, estabelecendo como será realizada sua execução. Ele também atua como um programa supervisor que estabelece uma camada de controle entre o hardware do computador e as aplicações de usuários. Uma de suas funções é estabelecer uma interface de software uniforme entre o computador, outros programas do sistema e programas de aplicação de usuários. Outra função fundamental de um sistema operacional é gerenciar os recursos de um computador de forma a promover sua eficiente utilização. Exemplos de sistemas operacionais são MS-DOS, Windows, OS/2, Linux, FreeBSD e Solaris — estes três implementações do sistema operacional Unix.

Nas próximas seções, os demais programas que complementam a tarefa do compilador e realizam as transformações necessárias em um código para que ele possa ser executado sob o controle do sistema operacional são apresentados de forma breve.

7.1 Interação do compilador com outros aplicativos

Antes de apresentar cada programa individualmente, deve estar claro que a invocação desses aplicativos ocorre de forma quase transparente para o usuário do compilador. Na verdade, o que ocorre é que os aplicativos que invocam o compilador atuam como um gerente que já invoca, conforme necessário, os demais aplicativos até a produção do programa executável. Como exemplo, considere como o compilador `g++` realiza, além da compilação propriamente dita, a integração com outros aplicativos do sistema.

O primeiro aplicativo invocado pelo g++, antes mesmo de iniciar a compilação, é o programa cpp, o pré-processador C. O pré-processador tem a tarefa de interpretar as diretivas iniciadas com o símbolo # na primeira coluna. Tais comandos não fazem parte da linguagem C++ que será analisada e, portanto, devem ser substituídos. Por exemplo, quando a diretiva #include é encontrada pelo pré-processador, todo o conteúdo do arquivo indicado pela diretiva é incorporado ao código-fonte. Assim, definições e declarações que estejam presentes nesse arquivo serão reconhecidas pelo compilador. Da mesma forma, definições associados às diretivas #define, bastante utilizadas na programação C, são armazenadas e, quando referenciadas no corpo do programa, são substituídas por sua expansão. Outra tarefa realizada pelo pré-processador é a exclusão de comentários do programa em código-fonte.

Após o pré-processamento, o compilador tem um arquivo que contém apenas instruções na linguagem (no caso, C++) e pode realizar a compilação. O resultado dessa compilação é um arquivo em linguagem simbólica. Se o programador precisar manipular esse arquivo, ele pode instruir o compilador para interromper o seu processamento nesse ponto e gerar como resultado apenas esse arquivo. Para tanto, a opção -S é utilizada e o arquivo gerado terá o mesmo nome-base do arquivo original mas com extensão .s. Por exemplo, o programa hello.cpp apresentado como exemplo na Seção 1.4.1, compilado tendo por alvo um processador da família Intel, produz com essa opção o seguinte arquivo:

```
        .file   "hello.cpp"
        .text
        .align 2
.LCFI2:
        movl    %eax, -4(%ebp)
        movl    %edx, -8(%ebp)
        cmpl    $1, -4(%ebp)
        jne     .L5
```

Há, nesse arquivo, uma combinação de diretivas para o programa montador, nas instruções iniciadas com ponto, e de instruções para o processador que deverá executar o programa. É um arquivo em formato texto, que pode ser analisado ou modificado pelo programador.

Se o processo não é interrompido nesse ponto, então o compilador prossegue e, com o uso do programa montador, gera o módulo objeto. Também é possível instruir o compilador para interromper o processo nesse ponto, por meio da opção -c. Nesse caso, o arquivo binário gerado contendo o módulo objeto tem extensão .o.

Finalmente, o compilador invoca o programa ligador para realizar a integração do código da aplicação do usuário com outros códigos. Para as linguagens C e C++, há uma série de recursos implementados em códigos externos que são considerados parte de um padrão, que deve estar disponível em qualquer plataforma com um compilador dessas linguagens. Para usar esses recursos, o programador não precisa passar nenhuma instrução especial para o ligador, pois o compilador sabe onde encontrá-los e já passa essa informação internamente. Para outros, o programador precisa passar explicitamente a instrução para o ligador ao invocar o compilador.

Considere o seguinte fragmento de um exemplo, que faz uso de uma rotina utilizada para apresentar erros de carregamento de bibliotecas dinâmicas:

```
#include <dlfcn.h>
#include <iostream>
using namespace std;
int main() {
  cout << dlerror() << endl;
}
```

Ao tentar compilar sem a instrução para o ligador, a seguinte mensagem é apresentada para o programador:

```
/tmp/ccmOnhhC.o: In function 'main':
...cpp:(.text+0x84): undefined reference to 'dlerror'
collect2: ld returned 1 exit status
```

Pode-se observar que a mensagem faz referência a um módulo objeto que está em um arquivo temporário e a uma localização do erro no código objeto — ou seja, refere-se a um erro após a compilação propriamente dita. Há a indicação da causa do erro: uma referência não definida para `dlerror`. Na última linha, finalmente, há a indicação de quem foi responsável pela geração da mensagem: o programa `ld`, o ligador da plataforma Linux.

Para resolver esse problema, é preciso indicar ao ligador — por intermédio do aplicativo `g++` — onde ele pode localizar a rotina desejada. No caso, essa rotina está num arquivo de biblioteca, `libdl.so`, na área padrão do sistema. Para passar a informação ao ligador, a chave `-ldl` (`l` de *library*, `dl` por ser o sufixo após `lib` no nome do arquivo) é usada no momento de invocar o compilador.

Com essas informações, o compilador pode transparentemente dirigir a execução desses programas e assim gerar o código pronto para a execução.

Na seqüência, montadores, carregadores e ligadores são apresentados.

7.2 Montadores

O montador é o programa do sistema responsável por traduzir um programa expresso em linguagem simbólica, em formato textual, para o código equivalente em linguagem de máquina, em formato binário. Assim, sua tarefa é transformar cada instrução do programa em linguagem simbólica para a seqüência de bits que codifica a instrução de máquina. Como cada processador tem seu próprio jogo de instruções e, conseqüentemente, sua própria linguagem simbólica e de máquina, montadores são desenvolvidos especificamente para cada plataforma-alvo.

Em geral, os montadores oferecem facilidades além da simples tradução de código em linguagem simbólica para código de máquina. Além das instruções do processador, um programa-fonte para o montador pode conter diretivas ou pseudo-instruções. Tais pseudo-instruções estão associadas ao montador e não ao processador para o qual ele gera código. São facilidades para simplificar o trabalho do programador, seja ele um ser humano, seja o compilador.

Outra facilidade usual em montadores é a possibillidade de definir macroinstruções, uma seqüência de instruções que será inserida no código ao ser referenciada pelo nome — similar ao que o pré-processador C faz com a diretiva `#define`. Um montador que suporte a definição e a utilização de macroinstruções é usualmente denominado macromontador (ou *macro-assembler*). Um montador multiplataforma (*cross-assembler*) é um montador que permite gerar código para um processador-alvo diferente daquele no qual o montador está sendo executado.

Na seqüência apresentam-se brevemente as atividades e estruturas de dados relacionadas ao processo de montagem, partindo da descrição do formato de entrada esperado até a geração do módulo-objeto de saída.

7.2.1 A estrutura de programas em linguagem simbólica

O montador recebe como entrada um arquivo texto cujas linhas são instruções em linguagem simbólica. Ao contrário do que ocorre na maior parte das linguagens de programação de alto nível, o formato de arquivos contendo código simbólico para um processador é pouco flexível. Embora o formato específico de um arquivo-fonte em linguagem simbólica possa sofrer ligeiras variações de acordo com o sistema, a descrição a seguir cobre a maior parte dos aspectos relevantes para a operação do montador.

O seguinte trecho de código apresenta um típico programa em linguagem simbólica com instruções de um processador da família 68K:

```
POS       DS.W      1
; Busca 0 na sequencia de inteiros
; DATUM definido alhures
SRCH0     MOVEA.L   #DATUM,A0
          MOVE.L    #DATUM,D0   ; guarda inicio
          CLR.W     D1
LOOP      CMP.W     (A0)+,D1
          BNE       LOOP
          SUB.L     A0,D0
          MOVE.W    D0,POS
          RTS
          END
```

Cada linha desse programa pode conter instruções ou comentários; uma linha é de comentário quando contém no início o caractere `;` (ponto-e-vírgula). Quem define esse padrão é o projetista do montador; alternativamente, ele poderia ter escolhido outro caractere para indicar o início de comentário, como `#` ou `*`.

As linhas de instrução contêm até quatro campos: rótulo, operação, operando e comentário.

A primeira coluna pode apresentar um rótulo opcional. Em programas em linguagem simbólica, destinos de desvios — sejam por chamadas de sub-rotinas, sejam por instruções de desvio — são referenciados pelo endereço em memória. A função básica do rótulo é criar uma identificação para poder referenciar simbolicamente, por meio de um nome em vez de um endereço, a linha de código rotulada. O montador pode reconhecer rótulos pela presença de caracteres distintos do caractere de comentário na primeira posição da linha.

A segunda coluna contém o campo de operação, que especifica a instrução que será montada. A operação pode ser tanto uma instrução de máquina, a exemplo de `MOVE` e `RTS`, como uma pseudo-instrução, como `DS`. No caso dos processadores da família 68K, pode ser necessário diferenciar, para uma mesma instrução, o tamanho do operando — byte (um byte), word (dois bytes, o padrão) ou long word (quatro bytes). Nesse exemplo de montador, os sufixos indicam o tamanho do operando — `.B`, `.W` e `.L`, respectivamente. Se for omitido, o tamanho *word* é assumido.

Dependendo da instrução presente na segunda coluna, o montador sabe se deve esperar zero, um ou dois operandos na terceira coluna, que corresponde

ao campo de operandos. Operandos podem fazer referências a registradores do processador (no caso do 68K, registradores de dados `D0` a `D7`, registradores de endereços `A0` a `A7`, registrador contador de programas `PC` e, para algumas instruções, registrador de códigos de condição `CCR`), a símbolos definidos pelos programas por meio de rótulos e a valores constantes.

A especificação do valor de uma constante que representa um operando imediato, indicado pelo prefixo `#`, pode se dar sob diversas formas de representação, também a critério do projetista do programa montador. Por exemplo, nas instruções

```
MOVE.B    #48,D0
MOVE.B    #$30,D0
MOVE.B    #@60,D0
MOVE.B    #%110000,D0
MOVE.B    #'0',D0
```

o operando imediato tem sempre o mesmo valor, representado, respectivamente, como um número decimal (sem prefixo adicional), hexadecimal (prefixo `$`), octal (prefixo `@`), binário (prefixo `%`) e ASCII (entre aspas simples). Qualquer que fosse a forma selecionada, o código de máquina gerado para essa instrução seria o mesmo:

```
00010000 00111100
00000000 00110000
```

Seqüências de caracteres (*strings*) são definidas também entre aspas simples. Por exemplo, `'ABC'` define uma seqüência de três bytes com valores $41, $42 e $43.

A quarta e última coluna, também opcional, corresponde ao campo de comentários. No exemplo, cada comentário é iniciado pelo caractere `;`, após o qual todo o restante da linha pode ser ignorado pelo montador.

Com o uso de pseudo-instruções e rótulos, é possível fazer referências a posições de memória e a variáveis por meio de identificadores simbólicos. Isso permite que o gerador de código ou o programador possa usar esses identificadores simbólicos como operandos de suas instruções sem ter de necessariamente saber a qual posição de memória a variável ou instrução está alocada. A regra para a composição de tais identificadores pode apresentar diferenças entre montadores distintos. Em geral, identificadores podem incluir letras minúsculas ou maiúsculas, dígitos e o caractere _ mas não podem ser iniciados por um dígito.

A definição de um módulo objeto contém pelo menos uma seção contendo o código de máquina (segmento de texto ou de instruções) e outra com os dados associados (segmento de dados). Além desses dois tipos de seções, um programa-fonte em linguagem simbólica pode conter uma seção de definições, usada para auxiliar na descrição dos programas sem produzir efeitos no código gerado.

Por conveniência de organização e da leitura do código-fonte, a seção de definições é tradicionalmente alocada no início do código. Assim, quando o código for lido por um ser humano, ele terá noção do significado das constantes simbólicas usadas ao longo do programa. Com relação aos segmentos de dados e de instruções, não há um posicionamento fixo. Na prática, um programa pode ter vários segmentos associados.

Para o montador, o posicionamento dos diferentes trechos de programa no código-fonte não é relevante. Na verdade, é necessário que o montador seja capaz de manipular representações simbólicas antes que elas tenham sido definidas. Considere o seguinte exemplo de um trecho de programa:

```
START    ADD.L    D0,D1
         JMP      NEXT
LOOP     ADD.L    #1,D1
NEXT     CLR.L    D5
         JMP      LOOP
```

Na segunda linha desse trecho de programa há uma referência a um símbolo, `NEXT`, cujo valor ainda não havia sido determinado — essa definição só acontecerá na quarta linha. Há duas possibilidades de lidar com essas referências futuras.

A primeira possibilidade é deixar uma lacuna reservada no código gerado associada ao operando da instrução da segunda linha. Posteriormente, quando houver uma definição desse valor — provavelmente quando o fim do arquivo com o código fonte for alcançado — essa lacuna será preenchida. Nesse caso, será possível gerar o código de máquina realizando um único passo (uma única leitura) sobre o arquivo. Entretanto, haverá uma complexidade maior de implementação do montador, que deverá manter referências a todas as lacunas que devem ser preenchidas ao final da montagem.

A outra possibilidade, conceitualmente mais simples, é realizar o processo de montagem em dois passos. O primeiro passo simplesmente lê o arquivo com o objetivo de criar uma tabela de símbolos, ou seja, obter os valores associados a todas as constantes simbólicas definidas no programa. No segundo passo, uma

nova leitura sobre o arquivo é realizada para gerar o código de máquina; nesse passo, a informação da tabela de símbolos criada no primeiro passo é utilizada.

Definições associadas ao código

Além das instruções do processador, um programa em linguagem simbólica preparado para um montador também pode conter pseudo-instruções que estabelecem a conexão entre referências simbólicas e valores a serem efetivamente referenciados. Cada montador pode oferecer um conjunto de pseudo-instruções diferenciado. As pseudo-instruções descritas a seguir constituem um subconjunto representativo de facilidades oferecidas por montadores.

A pseudo-instrução de substituição simbólica, `EQU`, associa um valor definido pelo programador a um símbolo. Por exemplo, a linha de instrução

```
SIZE      EQU      100
```

associa o valor decimal 100 ao símbolo SIZE, que pode ser posteriormente referenciado em outras instruções, como em

```
          MOVE     #SIZE,D0
```

Para essa pseudo-instrução, o rótulo deve estar sempre presente e o operando pode ser qualquer expressão que, quando avaliada, defina o valor para o símbolo. Essa expressão pode envolver outros símbolos já definidos.

A pseudo-instrução `EQU` define um símbolo que será usado durante o processo de montagem, mas que não fará parte do módulo-objeto nem tampouco corresponderá a variáveis na área de dados. Para definir constantes para a execução do código, ou seja, que ocuparão algum espaço em memória durante a execução do programa gerado, as pseudo-instruções `DC` e `DS` devem ser usadas.

A definição de variável inicializada, isto é, com algum valor constante definido no momento da alocação de espaço para a variável dá-se com o uso da pseudo-instrução `DC`, como nos exemplos

```
CONTADOR  DC.L     100
ARR1      DC.W     0,1,1,2,3,5,8,13
MENSAGEM  DC.B     'Alo, pessoal!'
```

O rótulo deve estar presente nessa instrução para permitir referenciar a posição de memória de cada variável ao longo do código. O sufixo no código de operação indica o tamanho em bytes para cada variável, seguindo no exemplo o padrão das instruções da família 68K. Assim, `CONTADOR` fará referência a uma

posição de memória cujos quatro bytes seguintes terão inicialmente a representação para o valor 100. ARR1 é uma referência para a posição de memória que dá início a um bloco contíguo de oito palavras de dois bytes, cada uma delas com o valor especificado na posição correspondente no operando. De forma similar, MENSAGEM faz referência ao início de um bloco de treze bytes representados em ASCII no operando.

Outra forma de reservar um espaço de memória para armazenar valores é com a pseudo-instrução de declaração de variáveis, DS, que reserva a quantidade de espaço indicada mas não inicializa seu conteúdo. Por exemplo,

```
VALUE    DS.W      1
```

associa ao símbolo VALUE uma referência para um endereço de memória que tem espaço suficiente para armazenar valores de uma variável com dois bytes de tamanho (word).

A pseudo-instrução ORG determina a origem do segmento. Um segmento é um conjunto de palavras de máquina que deve ocupar um espaço contíguo na memória principal. A posição de memória (endereço) associada ao início do segmento é denominada origem. O módulo-objeto gerado pelo montador contém tipicamente pelo menos dois segmentos, um segmento de código de máquina e um segmento de dados. O efeito dessa pseudo-instrução depende do tipo de montador que irá interpretá-la. Em alguns casos, pode ser uma definição de um endereço absoluto de memória no qual a origem do segmento deve ser posicionada. Em outros, é apenas a definição de um nome para referências futuras ao segmento quando sua posição de origem for definida. A forma genérica aqui adotada para a instrução será

```
         ORG ident
```

na qual ident é um identificador que pode ser um valor constante já definido, no caso dos montadores para carregadores absolutos como descrito na Seção 7.2.3, ou estar sendo definido como o nome de um segmento (nos demais casos).

Por exemplo, no trecho de programa a seguir

```
SEG1     EQU       $1000
         ORG       SEG1
         MOVE.W    DATA,D0
         MOVE.W    D0,DATA+2
         RTS
```

a pseudo-instrução `ORG` indica que a primeira instrução `MOVE.W` (na terceira linha) estará alocada à posição $1000 da memória, dando início ao segmento de código do módulo-objeto que será gerado.

Uma outra pseudo-instrução importante é `END`, que indica ao montador o fim do programa em linguagem simbólica. Seu formato geral é

```
END        ident
```

onde o identificador opcional `ident` no operando está associado a um rótulo do início do programa que está sendo encerrado por essa pseudo-instrução. Assim, esse identificador só deve estar presente uma única vez no código, mesmo que o código-fonte do programa esteja distribuído entre diversos arquivos — em geral, está associado a um "módulo principal". Nos demais módulos, a pseudo-instrução `END` aparece sem argumentos, sempre na última linha.

No caso de um programa cujo código-fonte está distribuído entre diversos segmentos, pode ser preciso fazer referências desde um segmento a variáveis definidas em outros arquivos-fontes. Para possibilitar essa conexão de referências, a pseudo-instrução `GLOB` é usada para indicar que cada um dos símbolos indicados pode ser referenciável externamente, ou seja, torna o símbolo visível globalmente. Seu formato genérico é

```
GLOB       idents
```

onde `idents` é uma lista de identificadores (separados por vírgulas, se mais de um estiver presente) definidos nesse segmento. Esses símbolos podem ser referenciados a partir de outros módulos, enquanto os demais símbolos definidos no segmento são considerados de escopo local, ou seja, são invisíveis para os módulos externos.

Alguns montadores definem pseudo-instruções tais como `EXTERN` para indicar que o símbolo que está sendo usado no módulo foi definido externamente, em outro módulo. No entanto, essa pseudo-instrução é desnecessária se for assumido que todos os símbolos referenciados mas não definidos localmente devem estar definidos externamente; esse comportamento é adotado pelo montador GNU, o programa `as`.

Macroinstruções

Uma macroinstrução é uma definição de um grupo de instruções que pode ser usado como se fosse uma única instrução ao longo do código-fonte. O uso de macros facilita a especificação de trechos repetitivos de código, que podem ser

invocados pelo programador como um única linha no programa. Por esse motivo, diversos montadores apresentam extensões com funcionalidades para a definição e utilização de macros.

Na sua forma mais simples, uma macro é simplesmente uma abreviatura para um grupo de instruções. A forma geral de definição de uma macro é

```
nome     MACRO      [argumentos]
         corpo
         ENDM
```

A pseudo-instrução `MACRO` marca o início da definição da macroinstrução. Toda macroinstrução tem um nome, especificado como o rótulo da pseudo-instrução e que será utilizado pelo programador para invocar a macro, e um corpo, que será usado pelo macromontador para substituir o nome usado pelo programador pela seqüência de instruções nele especificados. A pseudo-instrução `ENDM` marca o fim da definição.

Uma macro pode ter argumentos, usados para adaptar a expansão do corpo da macro. Assim, a seqüência de instruções especificadas no corpo da macro pode ser parametrizada pelos argumentos. Nesse exemplo, os parâmetros formais na definição de uma macroinstrução são precedidos pelo símbolo `&`.

A definição da macro `TOLOWER`, apresentada no exemplo a seguir, usa dois argumentos. O primeiro, `&IN`, é interpretado como a referência a um endereço de memória de um byte cujo conteúdo será copiado para o registrador `D0`, onde o sexto bit será setado. O conteúdo resultante será copiado para a posição indicada pelo segundo argumento, `&OUT`:

```
TOLOWER  MACRO      &IN,&OUT
         MOVE.B     &IN,D0
         ORI.B      32,D0
         MOVE.B     D0,&OUT
         ENDM
```

Uma vez que uma macro esteja definida, seu nome pode ser utilizado como se fosse uma operação do montador. A associação entre os argumentos da invocação da macro e os parâmetros formais da definição é feita pela posição da variável na declaração e invocação, assim como ocorre em sub-rotinas nas linguagens de alto nível.

Considerando essa definição, o uso da macro `TOLOWER` dar-se-ia como em

```
SIZE     EQU        5
CHARS_I  DC.B       'EA876'
```

```
CHARS_O  DS.B      SIZE
PROG001  MOVEA.L   #CHARS_I,A0
         MOVEA.L   #CHARS_O,A1
         MOVE.W    #SIZE,D0
LOOP     TOLOWER   (A0),(A1)
         ADDA.W    #1,A0
         ADDA.W    #1,A1
         DBF       D0,LOOP
         RTS
         END       PROG001
```

Após passar pela etapa de processamento de macros, esse código seria expandido para o seguinte código em linguagem simbólica:

```
SIZE     EQU       5
CHARS_I  DC.B      'EA876'
CHARS_O  DS.B      SIZE
PROG001  MOVEA.L   #CHARS_I,A0
         MOVEA.L   #CHARS_O,A1
         MOVE.W    #SIZE,D0
LOOP     MOVE.B    (A0),D0
         ORI.B     32,D0
         MOVE.B    D0,(A1)
         ADDA.W    #1,A0
         ADDA.W    #1,A1
         DBF       D0,LOOP
         RTS
         END       PROG001
```

Outra facilidade geralmente associada a macroinstruções é a possibilidade de definir expansões condicionais de trechos de código. Para tanto, uma pseudo-instrução como `AIF` pode ser definida. Assim, é possível isolar trechos da definição da macro que poderão ou não estar incluídos na respectiva expansão.

O formato dessa pseudo-instrução é

```
         AIF       cond  .mrot
```

sendo que `cond` é a condição que deve ser avaliada e `.mrot` é o rótulo de macro onde a expansão deverá continuar se a condição for verdade; caso contrário, a expansão continua na linha seguinte. A condição envolve tipicamente operadores relacionais de comparação de strings, aqui denotados `EQ` e `NE` para

expressar “igual a” e “diferente de”, respectivamente. O rótulo de macro é sempre iniciado por um ponto, uma forma de diferenciá-lo dos rótulos que serão incluídos no código expandido.

Para ilustrar esse conceito, considere novamente a definição da macro TOLOWER, que usa o registrador D0 para realizar a operação desejada. Se um dos argumentos para a macro for esse registrador, uma das instruções MOVE não deve ser incluída na sua expansão. Usando AIF, uma nova definição para essa macro que considera essa possibilidade é

```
TOLOWER  MACRO     &IN,&OUT
         AIF       (&IN EQ 'D0') .PULA
         MOVE.B    &IN,D0
.PULA    ORI.L     32,D0
         AIF       (&OUT EQ 'D0') .FIM
         MOVE.L    D0,&OUT
.FIM     ENDM
```

7.2.2 O processo de montagem

O processo de montagem de um código em linguagem simbólica pode apresentar pequenas diferenças em função das opções adotadas no projeto do montador, mas em linhas gerais as funcionalidades a seguir são suportadas.

Uma etapa inicial que pode ser suportada é o pré-processamento do código, onde informação não-relevante pode ser eliminada. Por exemplo, o pré-processador do montador as, disponibilizado nas distribuições do sistema operacional Linux, elimina comentários e converte constantes em formato de caracteres para as correspondentes constantes em valores numéricos. Na seqüência, o montador realiza o pré-processamento de macros, obtendo um código pronto para a criação do módulo objeto.

O montador receberá como entrada um arquivo em formato texto, do qual ele deverá ler cada linha para fazer o processamento que for necessário. Assim, um dos primeiros grupos de funcionalidades que se faz necessário é a manipulação de arquivos, descrita na Seção 1.3.1. Com essas funcionalidades é possível obter uma linha do arquivo com o código-fonte por vez.

Uma vez obtida a linha do arquivo, a tarefa de extrair de cada linha o campo de interesse estará representada por meio dos seguintes procedimentos, todos recebendo como argumento uma referência para a linha a ser processada:

GETLABEL(): extrai o rótulo da linha, se presente; caso contrário, retorna o valor nulo.

GETOPERATION(): extrai da linha o mnemônico de operação, que pode ser de uma instrução de máquina ou de uma pseudo-instrução.

GETOPERANDS(): obtém uma lista de operandos, que eventualmente pode ser vazia, a partir do conteúdo do campo de operandos da linha.

Processamento de macroinstruções

Se o montador admite a definição de macroinstruções no código-fonte, então o primeiro passo a ser realizado é o processamento das macrodefinições. De maneira geral, um processador de macro deve realizar quatro tarefas básicas:

1. Reconhecer as definições de macros;
2. Salvar as definições de macros de forma a possibilitar a posterior expansão;
3. Reconhecer as invocações a macros;
4. Expandir as invocações, possivelmente substituindo argumentos e verificando condições.

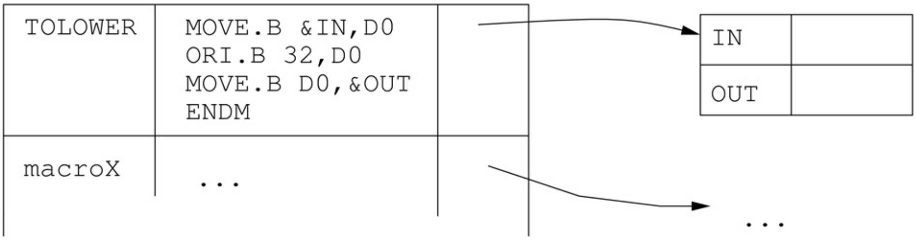
O processador de macros pode ser visto como um programa independente do montador, que é invocado antes do processo de montagem propriamente dito. Sua implementação mais simples pode ser realizada em dois passos.

No primeiro passo, cada linha do arquivo com o código-fonte (já sem comentários) é lida. Caso contenha na coluna do campo de operação a pseudo-instrução `MACRO`, então o que se segue é uma definição de macro, que deve ser armazenada. Uma estrutura de dados, a Tabela de Definição de Macro, é usada para guardar essas definições. A chave nessa tabela é o nome da macro, definido no campo de rótulo dessa mesma linha.

Associado a cada nome de macroinstrução, a tabela de definição de macro contém dois valores. Um valor é o corpo da definição da macro e o outro, a sua lista de parâmetros formais.

Para obter a lista de parâmetros formais, o processador de macro verifica se essa lista está presente no campo de operandos da linha. Se estiver, cada membro da lista será associado a uma entrada em outra estrutura de dados auxiliar, a Tabela da Lista de Argumentos, que será referenciada na tabela de definição de macro. Na Figura 7.2, a tabela de definição de macro está à esquerda e contém as referências para a tabela da lista de argumentos, à direita.

Figura 7.2 Estruturas de dados no processamento de macros



Para armazenar o corpo da macro, a linha a seguir é lida e copiada literalmente para a tabela. Verifica-se então se o campo de operação da linha copiada era `ENDM`; se for, então a definição dessa macro é concluída. Caso contrário, o procedimento é repetido para a linha seguinte.

O primeiro passo termina quando o processador de macro encontra a pseudo-instrução `END`, o que sinaliza o fim do código-fonte.

No segundo passo, cada linha de entrada é novamente lida e o campo de operação é obtido. Se a operação especificada for `MACRO`, essa linha e todas as que a seguem, até aquela que contenha a operação `ENDM`, são ignoradas. Caso contrário, verifica-se se a operação está presente na tabela de definição de macro. Se não estiver, a linha é copiada para o arquivo de saída na sua forma original. Caso contrário, a linha contém uma invocação de macroinstrução que deve ser expandida.

Na expansão da macro, verifica-se se a tabela da lista de argumentos contém algum elemento. Caso haja argumentos, as strings com os nomes dos argumentos são associadas, como valores na tabela, aos parâmetros, que são as chaves nessa tabela.

A expansão da macro continua pela leitura de linhas de código da definição, a partir da tabela de definição de macros. Caso a linha contenha no campo de operação a pseudo-instrução `ENDM`, o processo de expansão está concluído e continua a leitura do arquivo de entrada. Caso contrário, caso o campo de operando contenha algum nome iniciado pelo símbolo `&`, então esse nome é buscado na tabela da lista de argumentos para ser substituído pela string correspondente nessa expansão. Após essa substituição (se houver), a linha resultante é passada para o arquivo de saída.

O processamento de macros é concluído quando o processador de macro encontra a pseudo-instrução `END`, que é copiada para o arquivo de saída. Nele

haverá apenas linhas com instruções do processador ou pseudo-instruções; não há comentários, eliminados, ou macroinstruções, substituídas.

O processamento de macros pode ocorrer em um único passo caso se restrinja que todas as invocações a uma macro só podem ocorrer no código-fonte após sua definição, quando então os dois passos podem ser combinados.

Criação da tabela de símbolos

Criado o código em linguagem simbólica que contém apenas instruções de processador e pseudo-instruções, o passo seguinte é analisar as referências simbólicas contidas no código de forma a permitir a criação do módulo objeto. Nessa etapa do processamento, a atividade principal é a criação da tabela de símbolos do montador.

Na sua forma mais simples, a tabela de símbolos tem como chaves as strings com os nomes simbólicos definidos no programa em linguagem simbólica. Símbolos podem ser definidos como resultado de duas situações:

1. Como um rótulo em uma pseudo-instrução `EQU`; nesse caso, o valor do símbolo está definido no campo do operando.

2. Como um rótulo em uma outra instrução; nesse caso, o valor do símbolo está relacionado à posição de memória dentro do segmento onde ocorre a definição do símbolo.

Para poder criar a sua tabela de símbolos, o montador deve obter a informação sobre o espaço ocupado pelo código de máquina gerado por cada instrução do processador ou pseudo-instrução que tenha impacto na alocação de memória. Para tanto, o montador faz uso de duas estruturas auxiliares: a tabela de instruções de máquina e a tabela de pseudo-instruções.

A Tabela de Instruções da Máquina (ou MOT, de *machine operations table)* contém toda a informação necessária para permitir a tradução de um mnemônico para o código de máquina correspondente. A chave dessa tabela é o código de operação da instrução. Os valores incluem as regras para a geração do código de máquina e o espaço de memória que será ocupado pela instrução, em bytes. O conteúdo dessa tabela é determinado pelo processador para o qual o código está sendo gerado.

A Tabela de Pseudo-Instruções (ou POT, de *pseudo-operations table*) tem seu conteúdo definido pelos projetistas do montador. Assim como a MOT, é uma tabela com conteúdo exclusivamente para consulta, não sendo modificada durante a execução do programa. Tem como um dos valores as regras para obter

o espaço de memória que deve ser alocado em função da pseudo-instrução. Enquanto muitas das pseudo-instruções, tais como `EQU` ou `END`, não têm impacto sobre a ocupação de memória, `DS` e `DC` têm como efeito a necessidade de reservar ou modificar o conteúdo de posições de memória associadas ao programa. Assim, tais pseudo-instruções deverão gerar informação que irá fazer parte do módulo de carregamento gerado pelo montador.

A partir da informação derivada das tabelas do montador é possível saber quanto espaço de memória cada linha de instrução do código-fonte irá ocupar no módulo de carregamento gerado. Será a partir dessa informação que o montador poderá definir qual a posição a ser alocada para cada instrução do programa. Essa informação será mantida em uma variável contador de localização (LC, de *location counter*). Essa informação é transiente, ou seja, ela apenas existe durante a execução do montador.

7.2.3 Um montador em dois passos

Uma vez que o código-fonte em linguagem simbólica já tenha suas macroinstruções expandidas, a etapa de montagem propriamente dita pode ser iniciada. Para essa descrição, considera-se também que na etapa de pré-processamento os comentários foram eliminados do código-fonte. Assim, todas as linhas devem conter instruções simbólicas ou pseudo-instruções do montador.

Para introduzir os conceitos relacionados à montagem em dois passos, será inicialmente apresentado um exemplo motivador, apresentando um pequeno código em linguagem simbólica e o que seria gerado pelo montador a partir dele. Posteriormente, cada um dos passos do montador será detalhado.

Motivação

Considere como exemplo uma sub-rotina em linguagem simbólica do processador 68000 que deverá transferir um valor armazenado em uma posição para outra posição de memória, usando o registrador `D0` como armazenador temporário do dado. A sub-rotina terá o nome `PGM`, a posição de memória que tem o dado original será rotulada `VALUE` e a posição de memória destino será rotulada `RESULT`.

A listagem a seguir apresenta o código-fonte em linguagem simbólica, que é simplesmente uma seqüência de caracteres armazenada em um arquivo texto. Como descrito na Seção 7.2.1, strings na primeira coluna denotam rótulos para as posições de memória associadas; a segunda coluna contém strings que repre-

sentam as instruções (mnemônicos), e a terceira coluna contém strings representando os argumentos das instruções.

```
DATA     EQU      $6000
PROGRAM  EQU      $4000
         ORG      DATA
VALUE    DS.W     1
RESULT   DS.W     1
         ORG      PROGRAM
PGM      MOVE.W   VALUE,D0
         MOVE.W   D0,RESULT
         RTS
         END      PGM
```

No primeiro passo de execução, o montador deve gerar a tabela de símbolos. Para o exemplo anterior, a tabela de símbolos associada deve conter a seguinte informação:

Símbolo	**Valor**
DATA	$6000
PGM	$4000
PROGRAM	$4000
RESULT	$6002
VALUE	$6000

Há duas situações que devem ser consideradas para possibilitar a geração dessa tabela. A primeira situação, a mais simples, é quando a definição do símbolo é derivada de uma pseudo-instrução `EQU`. Esse caso é o mais simples porque toda a informação necessária para compor a entrada da tabela de símbolos está explícita na instrução. Assim foram definidos os símbolos `DATA` e `PROGRAM` na tabela de símbolos.

A segunda situação envolve a definição de símbolos usados como rótulos em outras instruções — no exemplo, o caso de `VALUE`, `RESULT` e `PGM`. Essa situação é mais complexa por necessitar de conhecimento da posição no segmento ou na memória que a instrução estará ocupando para poder efetivamente definir seu valor. Para tanto, o contador de localização (LC) deve ser atualizado durante o primeiro passo de acordo com o espaço reservado para cada instrução, informação que deve ser derivada a partir das tabelas de instruções de máquina (MOT) e de pseudo-instruções (POT).

No segundo passo do montador, o código deve ser efetivamente gerado. No caso desse exemplo, dois segmentos serão produzidos.

O primeiro segmento corresponde a uma área de dados, gerada a partir da interpretação das pseudo-instruções `DS` — eventualmente, a interpretação de pseudo-instruções `DC` também pode gerar informação para esse segmento, se estiver presente. O segmento gerado contém a seguinte informação:

Posição	$6000	$6002
Conteúdo	$0000	$0000

O segundo segmento corresponde à área de instruções, contendo o código de máquina associado a cada instrução. Para gerar esse código, o montador teve de *(i)* obter a codificação de máquina para cada instrução e *(ii)* resolver as referências simbólicas presentes nos operandos das instruções. Por exemplo, o código de operação para a primeira instrução é formado pela seqüência de dois bytes `3038`, em hexadecimal. Se os endereços são assumidos como de 32 bits, a referência após esse código de operação ocupa quatro bytes, `00006000`. A instrução seguinte tem o mesmo padrão, com um código de operação e uma referência a um endereço absoluto. Finalmente, a última instrução ocupa apenas dois bytes, pois não tem operandos. Desse modo, o segundo segmento contém:

Posição	$4000	$4002	$4004	$4006	$4008	$400A	$400C
Conteúdo	$3038	$0000	$6000	$31C0	$0000	$6002	$4E75

Na seqüência, serão analisados os procedimentos que o montador deve realizar para possibilitar essa geração de código.

Primeiro passo

Considerando a montagem em dois passos, o primeiro passo é responsável pela manipulação de rótulos de forma a descobrir o símbolo que está sendo criado em cada linha, se for o caso, e manter o controle sobre qual a posição (valor) de definição do símbolo. Essa última informação é obtida a partir da atualização do contador de localização LC a partir da avaliação do tamanho das instruções anteriores.

No primeiro passo do montador, o arquivo com o código-fonte é manipulado linha a linha. Para cada linha, o código de operação é analisado para descobrir se a instrução é uma pseudo-instrução (código de operação encontrado na POT) ou uma instrução em linguagem simbólica (código de operação encontrado na MOT). Caso o código não esteja em nenhuma das duas tabelas, o montador deve apresentar uma mensagem de erro para indicar que a instrução não foi reconhecida e o processo deve ser abortado.

Observe que duas buscas devem ser realizadas para cada linha de instrução obtida do código-fonte. Como a tabela de pseudo-instruções é em geral bem menor que a tabela de instruções de máquina, a busca é inicialmente realizada na primeira tabela. Apenas se o código buscado não for encontrado na POT a busca será realizada na tabela maior. A eficiência de implementação dessas buscas irá se refletir diretamente na eficiência do montador. Por esse motivo, é importante que bons algoritmos de busca e de manutenção da informação em tabelas sejam adotados na implementação do montador.

A tabela de símbolos é construída com a informação obtida do processamento das pseudo-instruções com código de operação `EQU`, `DC` ou `DS` (estas duas tratadas na condição *else* no caso *default*) ou com o processamento de instruções simbólicas com campo de rótulo não-nulo. Para a pseudo-instrução `EQU`, o rótulo é o nome do símbolo cujo valor deve ser obtido do operando. No caso das pseudo-instruções `DC` e `DS`, o rótulo é o nome do símbolo cujo valor é a posição corrente da instrução.

No processamento das demais pseudo-instruções, nenhum símbolo é criado. Se a pseudo-instrução é `ORG`, apenas o contador de localização deve ser atualizado. A pseudo-instrução `END` deve encerrar o primeiro passo do montador, invocando o segundo passo.

O processamento das pseudo-instruções `ORG` e `EQU` requer, já nesse passo, uma avaliação do valor do operando. Esse operando pode ser um literal ou um símbolo previamente definido. Caso seja um literal, o valor do operando é obtido a partir da conversão da string que representa o valor em alguma base — binária, octal, decimal ou hexadecimal. Caso seja um símbolo previamente definido, o valor é obtido a partir da busca da tabela de símbolos.

O procedimento do montador deve ainda avaliar o espaço ocupado pela instrução sendo processada, com o fim de atualizar corretamente o contador de localização. Para tanto, utiliza-se informação contida na tabela que contém a operação e o campo do operando.

Usando a rotina do exemplo anterior, é possível acompanhar o processo de atribuição de valores a símbolos que ocorre durante o primeiro passo da montagem. À medida que as linhas de códigos forem lidos, o contador de posição LC vai assumir os valores apresentados, em hexadecimal, na primeira coluna da Tabela 7.1.

O valor inicial do LC é 0. A pseudo-instrução `EQU` não ocupa espaço no código gerado e portanto não altera o valor do LC. O efeito das duas primeiras instruções é registrar na tabela de símbolos a informação que os símbolos `DATA` e `PROGRAM` têm os valores $6000 e $4000, respectivamente.

Tabela 7.1 Evolução do valor do contador de localização

LC	Instrução		
0000	DATA	EQU	$6000
0000	PROGRAM	EQU	$4000
0000		ORG	DATA
6000	VALUE	DS.W	1
6002	RESULT	DS.W	1
6004		ORG	PROGRAM
4000	PGM	MOVE.W	VALUE,D0
4006		MOVE.W	D0,RESULT
400C		RTS	
400E		END	PGM

A pseudo-instrução `ORG` da terceira linha também não ocupa espaço de código, mas tem efeito sobre o contador de localização — ele passará a registrar a posição de memória para a próxima linha do programa. Assim, LC passará a $6000, que é o valor de `DATA` obtido da tabela de símbolos.

A linha seguinte reserva espaço para uma variável. Na tabela de símbolos será registrado para o rótulo `VALUE` o valor de corrente do contador de localização, ($6000). Como o sufixo do tamanho para a pseudo-instrução `DS` é `.W`, será reservado espaço para uma *word* (dois bytes). Portanto, o LC é incrementado para o valor $6002. Da mesma forma, na linha seguinte, ao símbolo `RESULT` será associado o valor $6002 na tabela de símbolos e o contador de localização será incrementado para $6004.

Com a pseudo-instrução `ORG` da sexta linha o LC será alterado para $4000. Desse modo, ao rótulo definido na instrução seguinte, `PGM`, é associado na tabela de símbolos o valor $4000.

Para incrementar corretamente o LC, é preciso saber quantos bytes serão ocupados pela instrução da sétima linha. A partir do tratamento dos operandos e assumindo que endereços absolutos são representados em quatro bytes (*long word*), encontra-se a informação de que seis bytes serão usados para essa instrução, de forma que o LC será incrementado para $4006. Da mesma forma, encontra-se que a instrução seguinte também ocupa seis bytes, e o LC é incrementado para $400C.

Finalmente, no processamento da instrução `RTS` encontra-se que ela ocupa dois bytes, sendo que o valor do LC passa a $400E. A pseudo-instrução `END`

simplesmente indica ao montador o fim do código-fonte, tendo como argumento um endereço da primeira instrução executável.

Segundo passo

Uma vez que todos os símbolos que podem vir a ser usados como operandos das instruções no código-fonte já foram avaliados no primeiro passo do montador, é possível concluir a montagem. A atividade do montador no segundo passo é a geração do código de máquina.

A estrutura básica do segundo passo do montador é similar àquela do primeiro passo. O procedimento deve ler todas as instruções do código-fonte e, para cada linha, gerar o código de máquina correspondente. No processamento de operandos de cada instrução pode ser necessário realizar consultas à tabela de símbolos para obter os valores dos operandos simbólicos.

O procedimento preciso para o segundo passo do montador depende do tipo de carregador associado, mas a tarefa principal nesse passo é a produção do código de máquina associado à instrução e aos seus operandos. Tipicamente, na entrada da tabela correspondente à instrução sendo processada há uma referência para uma função que é capaz de realizar esse processamento. Por exemplo, o "código" gerado para a pseudo-instrução `DS` pode ser simplesmente uma seqüência de zeros no tamanho reservado pela instrução. Para a pseudo-instrução `DC`, o código gerado deve corresponder ao processamento dos literais e símbolos do operando. Para qualquer instrução simbólica do processador, essa função utiliza informação da MOT e o processamento dos literais e símbolos do operando para gerar o código de máquina correspondente.

Outra tarefa que é realizada é a escrita do módulo objeto. A forma usada pelo montador para organizar esse módulo de saída depende do modo de operação do carregador, como será visto nas próximas seções.

7.3 Carregadores e ligadores

Montadores geram como resultado um arquivo de conteúdo binário, contendo o código objeto associado ao arquivo-fonte de entrada. O arquivo com o código objeto contém parte da informação necessária à sua execução, mas para que ocorra a execução é preciso que esse código seja transferido para a memória principal. Se o código faz referências a elementos (dados ou rotinas) definidos externamente ao módulo, será preciso integrar essas referências ao código executável.

Neste capítulo, serão ainda descritas as atividades do sistema necessárias para que o programa montado possa efetivamente ser executado — a ligação, que resolve as referências que tenham sido feitas a dados e rotinas em outros programas, e o carregamento, que transfere o programa montado para a memória principal e dá início à sua execução.

7.3.1 Formato do módulo objeto

Com o processo de montagem, os segmentos do programa em linguagem simbólica são convertidos em arquivos no formato de módulo objeto, que serão posteriormente carregados para execução na memória. Tipicamente, um arquivo objeto contém os seguintes itens de informação:

Cabeçalho: contém a identificação do tipo de arquivo e dados sobre o tamanho do código e eventualmente o arquivo que deu origem ao arquivo objeto.

Código gerado: contém as instruções e dados em formato binário, apropriado ao carregamento.

Relocação: contém as posições no código onde deverá ocorrer mudanças de conteúdo quando for definida a posição de carregamento.

Símbolos: contém os símbolos globais definidos no módulo e símbolos cujas definições virão de outros módulos.

Depuração: contém referências para o código-fonte, tais como número de linha, nomes originais dos símbolos locais e estruturas de dados definidas.

Nem sempre todas essas informações precisam estar presentes no módulo objeto. Por exemplo, um arquivo em formato COM no sistema operacional DOS contém apenas o código gerado. Nesse caso, algumas restrições são impostas para garantir essa simplicidade. A posição de carregamento é predefinida no endereço `0x100` de algum segmento livre e o tamanho do código não deve exceder a capacidade de endereçamento interno a um segmento (64 KBytes). Caso o arquivo exceda esse tamanho, o programador será responsável por garantir a operação correta do programa executável.

7.3.2 Montagem e carregamento combinados

O esquema mais simples que incorpora a montagem e o carregamento como atividades separadas na execução de programas é o esquema absoluto. Nesse esquema, o montador gera um arquivo (módulo de carregamento) contendo, além

do código de máquina, a informação necessária para que o programa carregador possa carregar o código de máquina nas posições corretas de memória e transferir a execução para o programa carregado.

Esse tipo de esquema é na prática bastante limitado. No entanto, apresenta diversas funcionalidades que permitem introduzir detalhes importantes da operação de carregadores e montadores.

A forma mais elementar para executar o código de máquina gerado pelo montador é com o esquema *assemble and go*, no qual um único programa de sistema combina a realização das tarefas associadas a um montador e a um carregador. Nesse caso, não há a criação de um arquivo com o módulo objeto. Quando o código de máquina é gerado pelo montador, ele é colocado diretamente na posição de memória indicada pelo contador de localização. Assim, ao final da montagem, o programa executável já está na memória e o montador simplesmente transfere o controle de execução para a primeira instrução executável do código de máquina gerado.

Nesse tipo de esquema, no passo 2 do montador o código gerado é simplesmente copiado para a posição de memória indicada pelo contador de localização. Ao final da montagem, o montador simplesmente transfere a execução (com uma instrução de desvio incondicional) para o início do programa montado.

A grande desvantagem do esquema combinado está no fato de que cada execução do programa requer uma nova montagem, mesmo que o programa não tenha sido alterado. Outra desvantagem está no fato de que dois programas devem obrigatoriamente ocupar a memória principal, o montador e o programa montado. Assim, a utilização desse esquema está restrita a sistemas muito simples, não sendo de utilidade na prática.

7.3.3 Carregamento absoluto

Uma outra forma simples para contornar as desvantagens do esquema combinado consiste em separar o processo de montagem do processo de execução do código montado. Nesse caso, o montador gera um módulo de carregamento que não precisa ser regenerado a cada execução. Adicionalmente, as funcionalidades do montador não são necessárias para a execução — assim, o espaço de memória ocupado pelo programa montador pode ser liberado durante a execução do programa montado.

Para o carregamento absoluto o módulo de carregamento contém, além do código objeto, a informação sobre as posições de memória para as quais as linhas de código devem ser carregadas. Uma possível estratégia é associar um

registro do arquivo objeto a cada segmento, sendo que o início do registro indica a posição de carregamento e o tamanho do segmento em bytes.

O módulo de carregamento para o carregador absoluto é composto por dois tipos de registro. Todos os registros, exceto um, contêm informação que deve ser transferida para a memória na posição indicada (registro do tipo texto ou tipo 0, neste exemplo). O outro tipo de registro deve ter apenas uma ocorrência no fim do módulo de carregamento, correspondendo à informação do endereço para início da execução do programa (registro do tipo fim ou tipo 1, neste exemplo). Registros do tipo texto são gerados ao longo do segundo passo do montador, enquanto o registro do tipo fim é gerado ao final da execução do montador.

No exemplo do código gerado na Seção 7.2.3, a organização do módulo de carregamento segundo esse esquema seria composto por três registros:

0	00006000	4	00000000
0	00004000	E	30380000600031C0000060024E75
1	00004000		

O primeiro campo de cada registro indica o tipo do registro. O valor `0` nesse campo indica que o registro é do tipo texto e, portanto, o conteúdo a seguir (quarto campo), de dimensão quatro bytes (informação no terceiro campo), deverá ser transferido à memória a partir da posição \$6000 (informação do segundo campo). Similarmente, a informação do segundo registro indica a transferência dos 14 bytes do quarto campo a partir da posição \$4000. Finalmente, o último registro é do tipo fim, pois tem no primeiro campo o valor `1`. Dessa forma, o controle da execução deverá ser transferido para a posição indicada no segundo campo (\$4000).

As tarefas realizadas pelo carregador são simples. É necessário obter algumas informações de cada registro — tipo do registro, endereço inicial de carga para o conteúdo do registro, obter quantos bytes deverão ser transferidos e obter o conteúdo a ser transferido. Também é necessário transferir código do programa para a posição especificada de memória e dar início à execução, ao transferir o controle da execução para o endereço especificado.

7.3.4 Relocação e ligação

Os esquemas de montagem e carregamento absolutos, por sua simplicidade, não apresentam a flexibilidade necessária ao uso em sistemas operacionais modernos. Uma forte limitação está no fato de que o programador deve ter acesso direto a posições de memória, ao especificar exatamente em que região da memória o programa e seus dados serão carregados com a pseudo-instrução `ORG`.

Em sistemas operacionais modernos, tal limitação inviabiliza o uso daqueles esquemas. A memória é um recurso controlado pelo sistema, sendo que o programador não deve estar amarrado a conhecer posições da memória física para que o seu programa funcione corretamente. Por outro lado, desenvolver um programa completamente independente de sua localização é uma atividade complexa, embora possível. A solução é deixar que o software de sistema resolva problemas relacionados com posicionamento do código por meio da relocação.

Outro recurso que também requer a colaboração do montador e do carregador para seu funcionamento é a combinação, ou ligação, de módulos interdependentes mas montados independentemente. Nesse caso, deve ser possível a partir de um módulo fazer uma referência a um símbolo definido em outro módulo. No esquema de montador absoluto apresentado, tal situação geraria uma condição de erro pelo símbolo não estar definido, ou seja, não ter um endereço associado. Qualquer referência a símbolos externos deveria ser resolvida manualmente pelo programador. Com esquema de montagem e carregamento ajustáveis, o montador recebe a informação de que um símbolo está definido em outro módulo ou de que um símbolo estará sendo referenciado por outro módulo. Essa informação é registrada junto ao módulo objeto para uso pelo carregador, que realiza a resolução desses símbolos entre os módulos envolvidos.

Estruturas de dados adicionais

Os dois tipos de ajustes que podem ocorrer no conteúdo do módulo objeto são:

relocação: ajuste interno ao segmento;

ligação: ajuste entre segmentos distintos.

A atividade de relocação é realizada conjuntamente por montadores e carregadores. Montadores são encarregados de marcar as posições no código objeto passíveis de alteração devido à relocação do código. Carregadores devem reservar um espaço na memória de tamanho suficiente para receber o código de máquina e atualizar suas posições alteráveis a partir da informação sobre sua localização na memória.

No exemplo da rotina para transferência de valores, apresentado na Seção 7.2.3, as palavras que começam nas posições $4002 e $4008 do código objeto contêm endereços relocáveis. Verificando o código gerado, observa-se que a posição $4002–$4003 contém uma referência ao endereço $6000, e a posição $4008–$4009 contém uma referência ao endereço $6002. Se o início do segmento de dados for alocado a outro endereço de memória que não $6000, o

conteúdo dessas posições de memória deverá ser ajustado de acordo com essa mudança. O programa carregador é o encarregado de realizar esses ajustes. Para tanto, o módulo objeto deverá conter informação adicional que permita a realização dos ajustes.

Outro tipo de informação que deverá ser mantida no módulo objeto diz respeito a referências aos símbolos externos. Nesse caso, há duas situações que podem ser tratadas:

1. o símbolo é referenciado nesse segmento, mas é definido em outro segmento;

2. o símbolo é definido nesse segmento e poderá ser referenciado em outro segmento.

A primeira situação é usualmente descrita como uma referência externa (ER), enquanto a segunda situação será descrita como uma definição local (LD) de um símbolo externamente referenciável. A informação sobre esses dois tipos de símbolos deverá estar presente no módulo objeto.

Na seqüência, um esquema básico de resolução de relocação e referências externas — com carregadores de ligação direta — é apresentado. Para esse tipo de carregadores, o montador deverá incluir no módulo objeto estruturas de dados adicionais que incluam a informação necessária. São elas:

Dicionário de Símbolos Externos (ESD): contém todos os símbolos que podem estar envolvidos no processo de resolução de referências entre segmentos: símbolos associados a referências externas (ER), a definições locais (LD) ou a definições de segmentos (SD);

Diretório de Relocação e Ligação (RLD): para cada segmento indica que posições deverão ter seus conteúdos atualizados de acordo com o posicionamento deste e de outros segmentos na memória.

Essas duas estruturas de informação deverão estar presentes no módulo objeto. A partir delas, o carregador de ligação direta deve ser capaz de definir os valores para todos os símbolos com referências entre segmentos e reajustar o conteúdo das posições afetadas pela relocação.

O montador absoluto oferecia como resultado um módulo objeto com registros de dois tipos: registro com código de máquina (tipo 0) e um registro de fim (tipo 1). Um montador trabalhando no esquema de ligação direta deve fornecer dois tipos adicionais de registros além destes: um tipo para ESD e outro para RLD. Uma estrutura simplificada desses tipos de registros é indicada a seguir.

Registros do tipo ESD contêm todos os símbolos definidos dentro desse segmento que podem ser referenciados por outros segmentos, além de símbolos que são referenciados mas não definidos no segmento. Os símbolos locais que podem ser referenciados externamente podem ainda ser de dois tipos: definição do segmento ou definição local. Nos exemplos a seguir, um registro desse tipo apresentará a seguinte estrutura:

1. Tipo do registro (0).
2. Símbolo.
3. Tipo de definição (SD — segmento ou LD — local).
4. Endereço relativo no segmento.
5. Campo de dimensão, com o comprimento em bytes.

Nesse modelo simplificado de montagem e carregamento por ligação direta apresentado aqui, definições do tipo ER não receberão tratamento diferenciado. O campo de dimensão pode indicar tanto o espaço ocupado pelos dados de um símbolo (no caso de LD) como a dimensão total do segmento (no caso de SD).

Registros do tipo TXT contêm o código de máquina, com a informação do endereço relativo incorporada. O formato desse registro é:

1. Tipo do registro (1).
2. Endereço relativo.
3. Comprimento em bytes.
4. Código de máquina.

Registros do tipo RLD indicam quais posições no segmento deverão ter conteúdo alterado de acordo com os endereços alocados aos segmentos, indicando também a partir de que símbolo o conteúdo deverá ser corrigido. O formato desse registro adotado neste texto é:

1. Tipo de registro (2).
2. Posição relativa.
3. Comprimento em bytes.
4. Símbolo (base de ajuste).

Finalmente, um registro do tipo END especifica o endereço de início de execução para o segmento que contém a "rotina principal", sendo vazio para os demais segmentos:

1. Tipo de registro (3).
2. Endereço de execução.

7.3.5 Carregador de ligação direta

Programas de sistema primitivos usualmente combinavam a execução de diversas tarefas; por exemplo, no esquema combinado de montagem e carregamento, essas duas atividades eram executadas pelo mesmo programa. Do mesmo modo, as primeiras estratégias que incorporavam o processamento dos ajustes de ligação executavam essa tarefa juntamente com o processo de carregamento e os correspondentes ajustes de relocação.

Uma estratégia de ligação adotada em alguns sistemas foi o vetor de transferência, uma estrutura alocada no início da área de carregamento. Nesse vetor havia, para cada rotina, uma entrada cujo endereço era conhecido, pois estava no início da área de carregamento. Em cada entrada havia uma instrução de desvio incondicional para um endereço a ser definido, quando os endereços efetivos das rotinas fossem conhecidos. No código montado, a chamada de uma rotina usava o endereço do vetor de transferência e, de lá, era desviada para a posição efetiva de memória onde a rotina havia sido carregada. A limitação dessa estratégia é que apenas referências externas a rotinas podiam ser resolvidas; referências a variáveis externas deveriam ser tratadas manualmente pelo programador.

Esta seção apresenta as atividades desempenhadas por um carregador de ligação direta, outro esquema simples que combina carregamento e ligação em um único programa. Carregadores de ligação direta permitem a resolução a rotinas e dados externos, representando um avanço em relação ao esquema de vetor de transferência.

A operação do carregador de ligação direta será apresentada a partir de um exemplo simples. Considere o seguinte programa, que faz referência a um símbolo externo `DIGIT`:

```
MAIN     MOVE.B   DIGIT,D0
         CMPI.B   #10,D0
         BLT      ADD_0
         ADDQ.B   #('A'-'0'-10),D0
ADD_0    ADDI.B   #'0',D0
         MOVE.B   D0,CHAR
         RTS
CHAR     DS.W     1
         END      MAIN
```

Esse programa obtém um valor inteiro entre 0 e 15 de `DIGIT` e irá colocar na variável `CHAR` sua representação ASCII, entre ‘0’ e ‘F’.

No segmento onde `DIGIT` é definido, é preciso indicar que esse símbolo poderá ser referenciado externamente. Para tanto, a pseudo-instrução `GLOB` é utilizada.

O trecho a seguir ilustra a definição de `DIGIT` em outro segmento:

```
        GLOB      DIGIT
PGM     MOVE.W    VALUE,D0
        MOVE.W    D0,DIGIT
        RTS
VALUE   DS.W      1
DIGIT   DS.W      1
        END
```

O efeito da pseudo-instrução `GLOB` será a criação de um registro do tipo `ESD` com tipo de definição `LD` — quando a posição relativa do símbolo for definida na tabela de símbolos locais, a informação do registro deverá ser complementada.

O montador deverá gerar o seguinte módulo objeto (com campos separados por pontos) para o segmento `MAIN`:

```
0.'MAIN'.'SD'.00.1C
1.00.6.103900000000
1.06.4.0C00000A
1.0A.2.6D02
1.0C.2.5E00
1.0E.4.06000030
1.12.6.13C00000001A
1.18.2.4E75
1.1A.2.0000
2.02.4.'DIGIT'
2.14.4.'MAIN'
3.00
```

Nesse exemplo, valores numéricos são apresentados em hexadecimal e símbolos na forma de seqüências ASCII — na realidade, o módulo objeto teria apenas a seqüência de bits associada a cada uma dessas representações.

O início do módulo objeto contém o diretório de símbolos externos (ESD, registros com primeiro campo com valor 0), o código de máquina gerado (TXT, registros com primeiro campo 1), o diretório de relocação e ligação (RLD, registros com primeiro campo 2) e o registro de fim de segmento (END, com primeiro

campo 3). Para o registro de fim de segmento, a posição relativa de execução (posição 00) é especificada.

Similarmente, para o segmento `PGM` o seguinte módulo é gerado:

```
0.'PGM'.'SD'.00.12
0.'DIGIT'.'LD'.10.2
1.00.6.30390000000E
1.06.6.33C000000010
1.0C.2.4E75
1.0E.2.0000
1.10.2.0000
2.02.4.'PGM'
2.08.4.'PGM'
3.
```

O carregador de ligação direta recebe como argumentos a lista de módulos a carregar, trabalhando usualmente em vários passos — tipicamente dois.

Uma das atividades realizadas no primeiro passo é a alocação de espaço em memória. Para saber quanto espaço é necessário, a informação sobre o comprimento de cada segmento — presente em registros tipo ESD, com tipo de definição SD — é obtida. Uma vez determinado qual o endereço inicial de carregamento do segmento (IPLA — *Initial Program Load Address*), o carregador inicia a criação de uma Tabela de Símbolos Externos Globais (GEST). Para tanto, apenas a informação presente em registros do tipo ESD, com tipos de definição SD e LD, é utilizada. Na fase de definição da GEST, um possível erro que poderia ser detectado e indicado ao usuário é a duplicação na definição de símbolos na tabela, ou seja, um mesmo símbolo sendo redefinido em segmentos distintos.

No último passo sobre os arquivos de entrada, o carregador irá realizar a transferência do código de máquina para a memória e transferir o controle da execução do programa para o endereço inicial do programa recém-carregado. Nesse passo, o carregador volta a tomar o endereço inicial de carregamento, lendo novamente cada módulo objeto na seqüência original. Quando o registro lido é do tipo ESD, o único processamento envolvido é obter o comprimento do segmento de forma a permitir a atualização correta da variável que indica a posição inicial de carga de cada segmento. Essa informação está contida no registro ESD cujo tipo de definição é SD (*Segment Definition*).

Os registros do tipo TXT têm seu conteúdo transferido para a memória principal. Cada campo do registro — posição relativa ao início do segmento, ta-

manho e conteúdo — é obtido, sendo que o endereço de destino é resolvido tomando por base o valor do início do segmento.

Ao final da transferência, as posições indicadas em registros do tipo RLD têm seu conteúdo ajustado a partir da informação registrada na GEST. Os registros do tipo RLD têm a indicação da posição relativa que deve ser corrigida, sendo que a posição de memória cujo conteúdo será alterado é obtida a partir da combinação dessa informação com o endereço de início do segmento. O valor pelo qual o conteúdo deverá ser alterado é especificado pelo campo de símbolo presente nesse registro — o símbolo é lido do registro e seu valor é obtido a partir de uma busca na GEST. Nesse ponto, pode-se detectar um erro caso algum símbolo tenha sido referenciado e não definido em nenhum módulo.

7.3.6 Ligadores

A estratégia de ligação direta apresentada na Seção 7.3.5 ilustra bem o princípio de resolução de endereços entre módulos, mas apresenta limitações. Uma dessas limitações é que o programa carregador é mais complexo que o carregador absoluto, ocupando mais espaço em memória. Como o carregador compartilha memória com o programa sendo executado, menos memória é deixada para a aplicação.

Uma estratégia alternativa é isolar os procedimentos de ligação e de carregamento em programas separados. O programa ligador recebe como entrada os diversos módulos a conectar, gerando como saída um único módulo de carga. O programa carregador recebe o módulo de carga como entrada, transfere seu código para a memória e realiza apenas os ajustes de relocação de acordo com o endereço-base de memória.

A principal diferença na criação de um módulo objeto preparado para a ligação é que o endereço inicial de carga é considerado como sendo 0. Assim, todos os endereços passam a ser relativos ao início do módulo de carga. A informação para realizar os ajustes de relocação está presente no final do módulo de carga e indica quais posições de memória deverão ser atualizadas após alocação. Nesse caso, não é necessário manter o símbolo de referência para o ajuste de relocação, pois o endereço inicial de carregamento é a base para todos os ajustes.

A saída é enviada a um arquivo (o módulo de carga) em vez de colocada na memória. Outro programa, o carregador, será responsável por transferir o módulo de carga para a memória e realizar ajustes de relocação (mas não de ligação). O carregador obtém a informação de quanto espaço deve ser alocado do registro inicial (tipo 0), transfere o código (registros tipo 1) para a área de

memória alocada e usa a informação do diretório de relocação (registros tipo 2) para ajustar o endereço nas posições indicadas.

Um ligador que produz módulos de carga relocáveis é usualmente denominado *link-editor*, sendo que os mais elaborados permitem definir diversas seções e a inclusão de comandos específicos para a ligação. Um exemplo é o comando **ld** do Unix. A implementação desse comando (GNU) em Linux incorpora a seguinte informação na sua documentação:

> **ld** combina um número de arquivos objeto e de bibliotecas, reloca seus dados e amarra referências simbólicas. Usualmente o último passo na compilação de um programa é executar o **ld**.

Arquivos de entrada para o programa **ld** podem conter diretivas expressas em uma linguagem própria de comandos (*Linker Command Language*). A linguagem de comandos suportada por **ld** permite especificar para o processo quais são os arquivos de entrada, qual o formato de cada arquivo, qual deve ser o formato (*layout*) do arquivo de saída, quais são os endereços de seções e qual o posicionamento de blocos comuns.

7.3.7 Bibliotecas

Quando um programador usa em seus programas funções oferecidas pelo sistema, essas funções estão usualmente já montadas, disponibilizadas em formato objeto. Entretanto, em vez de ter um arquivo objeto para cada função (o que tornaria o número de objetos excessivo), essas funções estão usualmente organizadas na forma de arquivos do tipo biblioteca.

Bibliotecas são arquivos que contêm um conjunto de módulos objetos, normalmente agrupados de acordo com sua funcionalidade. A origem do termo “biblioteca” vem da época dos computadores de grande porte, para os quais as rotinas auxiliares eram mantidas em fitas ou cartões armazenados em salas com prateleiras.

Nesta seção, bibliotecas estáticas serão descritas; bibliotecas dinâmicas serão vistas na Seção 7.3.8. Uma biblioteca estática fornece código objeto que deve ser integrado ao módulo executável antes do momento de execução, durante o processo de ligação.

Em geral, o sistema operacional apresenta utilitários para manipular arquivos tipo biblioteca. Em Unix (Linux), o utilitário `ar` é utilizado para criar, manter e extrair módulos de arquivos de bibliotecas estáticas. Por exemplo, se um módulo objeto `arqmat.o` tiver sido criado com a linha de comando

```
> gcc -c arqmat.c
```

esse módulo pode ser incluído em uma biblioteca `libmy.a`, criada pelo usuário, com a linha de comando

```
> ar -r libmy.a arqmat.o
```

A chave `-r` indica que ocorrerá uma troca (*replacement*) do módulo, caso haja uma versão anterior já armazenada na biblioteca; caso contrário, o módulo é acrescentado à biblioteca.

Se essa for a primeira operação com essa biblioteca, ela será criada pelo programa `ar`. Nesse caso, uma solicitação de listar o conteúdo (com a opção `-t`) mostrará que apenas esse módulo está presente:

```
> ar -t libmy.a
arqmat.o
```

Novos módulos podem ser similarmente incluídos:

```
> ar -r libmy.a convexp.o
> ar -t libmy.a
arqmat.o
convexp.o
```

A estrutura típica de um arquivo do tipo biblioteca nesse tipo de sistema operacional é composta por uma identificação do tipo de arquivo, por um diretório de membros do arquivo e por uma seção com o conteúdo de cada membro do arquivo.

O diretório de membros do arquivo é composto por uma série de cabeçalhos de membro, sendo que cada cabeçalho de membro contém informações tais como o nome do membro e sua dimensão em bytes.

Por exemplo, bibliotecas estáticas no sistema operacional Unix (arquivos com extensão `.a`) têm a string de oito caracteres `!<arch>\n` como identificação no início do arquivo. O cabeçalho de cada um dos membros pode ser mapeado a uma estrutura em C com as seguintes informações:

```
char name[16];
char modtime[12];
char uid[6];
char gid[6];
char mode[8];
char size[10];
char eol[2];
```

A informação `name` contém o nome do membro, terminado com o caractere `/` e com o espaço restante preenchido com brancos. Sua dimensão está associada aos primeiros sistemas operacionais, que limitavam a quantidade de caracteres nos identificadores. Atualmente, para lidar com nomes de dimensão maior, esse campo pode conter um nome vazio seguido de uma referência a uma posição em uma tabela complementar de strings, onde nomes maiores são armazenados.

A data de modificação (`modtime`) é expressa como no sistema operacional, o valor decimal de segundos decorridos desde 1º de janeiro de 1970.

A identificação do usuário e grupo criador do módulo também segue o padrão do sistema operacional Unix. São números decimais que identificam cada usuário e grupo registrado no sistema.

As permissões de acesso para o módulo são representadas como um número octal que associa para cada nível (próprio usuário, outros usuários do mesmo grupo ou usuários externos) se há permissão de leitura, escrita e execução ou acesso para o membro.

O tamanho do módulo em bytes, expresso como um número decimal, é seguido pelo terminador de cabeçalho, usualmente a seqüência com os dois caracteres `\` e `n` para indicar um "fim de linha".

A estrutura de um arquivo do tipo biblioteca pode ser usada para agregar qualquer tipo de conteúdo, mas usualmente apenas módulos objetos são agrupados em bibliotecas.

No processo de ligação, além dos módulos objetos gerados a partir dos arquivos-fontes originais, o programador pode especificar arquivos do tipo biblioteca. Inicialmente, o ligador irá resolver as referências que puderem ser estabelecidas a partir dos módulos objetos fornecidos. Se, ao final dessa etapa, ainda houver referências não-resolvidas, o ligador procura pela definição dos símbolos dentro das bibliotecas. Ao encontrar o cabeçalho do módulo especificado, o ligador obtém dali toda a informação necessária para extrair apenas o módulo desejado e assim integrá-lo ao módulo de carga executável.

Arquivos de biblioteca são amplamente utilizados, embora nem sempre de forma explícita. Por exemplo, na implementação GNU para o compilador C a biblioteca `libc.a`, armazenada no diretório `/usr/lib`, contém as funções da biblioteca padrão da linguagem. Como essas funções são amplamente utilizadas (tal como a rotina `printf`), o programador não precisa explicitar para o ligador que essa biblioteca deverá ser utilizada para a resolução de símbolos — o próprio compilador `gcc` irá integrar essa biblioteca ao processo de ligação.

Quando uma outra biblioteca tiver de ser utilizada, contendo por exemplo rotinas matemáticas (em Unix, na biblioteca `libm.a`) ou rotinas associadas a

outros pacotes ou aplicativos (bancos de dados, interfaces gráficas), é preciso passar essa informação ao ligador. No caso do ligador `ld`, há duas chaves relacionadas ao fornecimento dessa informação — essas chaves podem ser especificadas para o compilador, que as repassa ao programa ligador. A chave `-l`*xxx* indica que a biblioteca cujo nome é `lib`*xxx*`.a` deve ser incorporada ao processo de resolução de referências.

Por exemplo, considere o seguinte programa que incorpora uma rotina matemática — no caso, `cos` para o cálculo do co-seno de um valor real especificado na linha de comando:

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  double valor, result;
  if (argc != 2) {
    fprintf(stderr, "%s requer um argumento numerico\n", argv[0]);
    return 1;
  }
  valor = atof(argv[1]);
  result = cos(valor);
  printf("Co-seno de %lf: %lf\n", valor, result);
  return 0;
}
```

Se o correspondente arquivo de biblioteca com rotinas matemáticas, que contém o código objeto para a rotina `cos`, não for especificado, um erro de ligação será gerado:

```
> gcc calccos.c
.../cKiW4b.o: In function 'main':
.../cKiW4b.o(.text+0x54): undefined reference to 'cos'
collect2: ld returned 1 exit status
```

Para incluir a ligação das rotinas matemáticas, o código deve ser compilado e ligado com a inclusão da chave `-lm`, como em

```
> gcc calccos.c -lm
```

A outra chave associada à especificação de bibliotecas é `-L`, que especifica um diretório onde arquivos do tipo biblioteca estarão armazenados, caso seja necessário fazer essa busca em um diretório distinto dos usados por padrão pelo sistema operacional — tipicamente, os diretórios `/lib`, `/usr/lib` e `/usr/local/lib`.

7.3.8 Carregamento e ligação dinâmicos

Os esquemas de ligação e carregamento apresentados até o momento assumem que o módulo executável, uma vez carregado a uma área da memória principal, será o "proprietário" dessa área até o fim de sua execução. Em sistemas multiusuários mais recentes, não é isso o que ocorre. Programas em execução podem ser retirados da memória (*swaped-out*) e depois retornar à memória (*swap-in*) em outra posição diferente daquela na qual estava executando inicialmente.

Para atender a esse tipo de necessidade, utilizam-se esquemas de ligação e carregamento dinâmico. O princípio básico desses esquemas é que referências a endereços (de dados ou de instruções) são mantidas na forma relativa até o momento em que eles são realmente necessários, ou seja, até o momento de execução da instrução que contém essa referência. Usualmente, essa funcionalidade deve ter parte suportada em hardware de modo a não haver degradações sensíveis de desempenho.

Há dois esquemas básicos de ligação dinâmica: em tempo de carregamento (*load-time*) ou em tempo de execução (*run-time*). Na ligação dinâmica em tempo de carregamento, o módulo de carga primário (módulo da aplicação) é inicialmente transferido para a memória. Qualquer referência nesse módulo para módulos externos (módulos-alvo) faz com que o carregador procure cada módulo-alvo, carregue-o para a memória e altere as referências para um endereço relativo em memória a partir do início do módulo da aplicação.

Entre as vantagens nesse esquema de carregamento pode-se destacar:

- facilidade de atualização de versões de módulo-alvo sem alterar a aplicação;
- facilidade no suporte ao compartilhamento de módulos-alvo entre aplicações distintas — se o sistema operacional detectar que um módulo-alvo já está em memória, uma única cópia pode ser mantida em memória (contanto que o conteúdo do módulo-alvo não seja alterado pela aplicação).

A diferença para a ligação dinâmica em tempo de execução está no fato de que o carregamento e a resolução de referências são retardados até o momento em que a instrução com a referência ao módulo externo é executada. As referências a módulos externos continuam presentes na aplicação, mas se em alguma execução a lógica de fluxo de controle fizer com que aquela referência não seja executada, então o módulo-alvo não será carregado à memória por aquela aplicação. As vantagens do esquema de ligação dinâmica em tempo de carregamento continuam válidas também nesse caso.

Em versões mais recentes do sistema operacional Linux, os módulos objeto e de carga estão principalmente em formato ELF (*Executable and Linking Format*). Bibliotecas para esse tipo de arquivos são denominadas bibliotecas dinâmicas ou arquivos de objetos compartilhados, que são diferenciadas das bibliotecas estáticas por sua extensão — estáticas têm extensão `.a` (*archive*) e dinâmicas têm extensão `.so` (*shared objects*).

Sob o ponto de vista do programador usuário, não há diferença no procedimento para uso de rotinas em bibliotecas estáticas ou dinâmicas — da mesma forma, rotinas em bibliotecas padrão são automaticamente buscadas e outras bibliotecas deverão ser especificadas com a chave `-l`. A diferença está na forma de operação interna do ligador e carregador, que utiliza a interface de programação (API) associada ao formato ELF.

ELF apresenta uma API que pode ser utilizada em programas do sistema desenvolvidos em C para manipular objetos em biblioteca compartilhadas durante a execução de um programa. Essas rotinas, disponibilizadas pela biblioteca `libdl.so`, são:

```
#include <dlfcn.h>
void       *dlopen (const char *filename, int flag);
const void *dlsym  (void *handle, const char *symbol);
int         dlclose(void *handle);
const char *dlerror(void);
```

A rotina `dlopen` disponibiliza uma biblioteca dinâmica para o programa em execução — em outros termos, as rotinas no arquivo especificado são mapeadas para o espaço de endereçamento do processo em execução. O seu valor de retorno é um ponteiro (*handle*), utilizado nas chamadas posteriores de manipulação da biblioteca. O argumento `flag` indica quando deverá se dar o carregamento. Se tiver o valor `RTLD_NOW` (uma constante definida no arquivo `dlfcn.h`), o carregamento deverá ser imediato, ou seja, ao retornar dessa rotina a biblioteca já estará carregada na memória. Caso o valor especificado seja `RTLD_LAZY`, o carregamento será postergado até o momento em que houver (e se houver) referência a um símbolo dessa biblioteca. Em caso de erro, o apontador nulo será retornado.

A rotina `dlsym` retorna o endereço do símbolo especificado (variável ou função) que está disponível na biblioteca compartilhada que foi aberta. Em caso de erro, o apontador nulo será retornado.

Quando a biblioteca compartilhada não é mais necessária, ela é liberada com a invocação da rotina `dlclose`, que retorna 0 em caso de sucesso.

Em qualquer situação de erro, a rotina `dlerror` pode ser invocada para obter uma string com o diagnóstico do erro.

O exemplo a seguir ilustra como a função `cos` pode ser dinamicamente carregada da biblioteca `libm.so` usando ELF em Linux:

```
#include <dlfcn.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  void *handle;
  double (*cosine)(double);
  char *error;
  handle = dlopen("/lib/libm.so.5", RTLD_LAZY);
  if (!handle) {
    fputs (dlerror(),stderr);
    return 1;
  }
  cosine = dlsym(handle, "cos");
  if ((error = dlerror()) != 0) {
    fputs(error, stderr);
    return 1;
  }
  printf("%f\n", (*cosine)(2.0));
  dlclose(handle);
}
```

Para criar uma biblioteca compartilhada, inicialmente é preciso gerar um módulo objeto que possa ser carregado dinamicamente. Para tanto, o código gerado deve ser independente de posição. O compilador `gcc` permite a criação desse tipo de código de forma automática, pelo uso da chave `-fPIC` (de *position-independent code*):

```
> gcc -fPIC -c arqmat.c
> gcc -fPIC -c convexp.c
```

Para criar a biblioteca dinâmica contendo os objetos compartilhados, o próprio compilador é utilizado:

```
> gcc -shared -o libmyd.so arqmat.o convexp.o
```

A chave `-shared` indica para o programa `gcc` que o programa que está sendo gerado (indicado pela opção `-o`) é uma biblioteca compartilhada cujos módulos podem ser carregados e ligados dinamicamente. Caso algum desses módulos faça referências a arquivos em outras bibliotecas dinâmicas, é possível

tornar transparente para o usuário a necessidade de se carregar essa outra biblioteca especificando-a no momento da criação. Por exemplo, se `arqmat.o` faz uso de rotinas em `libm.so`, a linha de comando

```
> gcc -shared -o libmyd.so arqmat.o convexp.o -lm
```

fará com que `libm.so` seja automaticamente carregada quando `libmyd.so` for especificada.

O padrão em sistemas operacionais modernos é utilizar arquivos compartilhados com carregamento e ligação dinâmica. O compilador `gcc` inclui a opção `-static` caso seja necessário criar um executável ligado estaticamente, mudando assim o comportamento padrão.

7.4 Exercícios

7.1 Para o seguinte programa em linguagem simbólica do processador 68000, apresente o conteúdo da tabela de símbolos gerada pelo montador. Assuma que o código de operação ocupa dois bytes com endereçamento direto para registradores e, para endereçamentos absoluto e imediato, quatro bytes adicionais são necessários.

```
DATA     EQU      $FA
PRGM     EQU      $100
         ORG      PRGM
SPIL     CLR.L    XYZ
         MOVE.L   (A7)+,D0
         BEQ      FSPIL
LOOP     MOVE.L   (A7)+,D1
         ADD.L    D1,XYZ
         SUB.L    #1,D0
         BNE      LOOP
FSPIL    RTS
         ORG      DATA
FOO      DS.W     1
XYZ      DS.L     1
         END      SPIL
```

7.2 Qual a diferença entre o resultado gerado por um montador absoluto e um montador de ligação direta?

7.3 Qual a diferença entre ajuste de relocação e ajuste de ligação?

7.4 Qual a diferença entre ligação dinâmica em tempo de carga e em tempo de execução?

7.5 Quando é interessante que a tabela de símbolos gerada por um montador seja incorporada ao módulo objeto gerado?

7.6 Como se comparam os tamanhos dos módulos objetos nos esquemas de carregamento estático e de carregamento dinâmico?

7.7 Um montador de ligação direta aplicado a dois arquivos em linguagem simbólica do 68K gerou os seguintes módulos objetos:

Módulo 1	**Módulo 2**
0.'MAIN'.'S'.0000.001A 0.'RESULT'.'L'.0018.0002 1.0000.06.203900000014 1.0006.06.4EB900000000 1.000C.06.33C000000018 1.0012.02.4E75 1.0014.04.00004E75 2.0002.04.'MAIN' 2.0008.04.'CALC' 2.000E.04.'MAIN' 3.02.0000	0.'CALC'.'S'.0000.0006 1.0000.02.2200 1.0002.02.9081 1.0004.02.4E75 3.00

Passados como argumentos nessa ordem (módulo 1 seguido de módulo 2) para um carregador de ligação direta, obteve-se o endereço inicial de carga (IPLA) \$0200.

(a) Qual o conteúdo da Tabela de Símbolos Externos Globais (GEST) gerada pelo carregador?

(b) O diagrama a seguir é um mapa de conteúdo da memória após o carregamento *sem* os ajustes de ligação e relocação. Indique nesse mapa quais posições são ajustadas pelo carregador e qual o novo conteúdo dessas posições.

Posição	**Conteúdo**
0200	2039
0202	0000
0204	0014
0206	4EB9
0208	0000
020A	0000
020C	33C0
020E	0000
0210	0018
0212	4E75
0214	0000
0216	4E75
0218	0000
021A	2200
021C	9081
021E	4E75

Referências Bibliográficas

[Aho et al., 1995] Aho, A. V., Sethi, R., and Ullman, J. D. (1995). *Compiladores: Princípios, Técnicas e Ferramentas*. LTC Editora. Tradução de *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[Aho and Ullman, 1977] Aho, A. V. and Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley.

[Backus, 1998] Backus, J. (1998). The history of FORTRAN I, II, and III. *IEEE Annals of the History of Computing*, 20(4):68–78.

[Daltrini et al., 1999] Daltrini, B. M., Jino, M., and Magalhães, L. P. (1999). *Introdução a Sistema de Computação Digital*. Makron Books.

[Donnelly and Stallman, 2006] Donnelly, C. and Stallman, R. (2006). *Bison: The Yacc-compatible Parser Generator*. Free Software Foundation.

[Donovan, 1972] Donovan, J. J. (1972). *Systems Programming*. McGraw-Hill.

[Eckel, 2000] Eckel, B. (2000). *Thinking in C++*. Prentice Hall.

[Grune and Jacobs, 1990] Grune, D. and Jacobs, C. (1990). *Parsing Techniques: A Practical Guide*. Ellis Horwood Limited.

[Hunter, 1981] Hunter, R. (1981). *The Design and Construction of Compilers*. John Wiley and Sons.

[Kernighan and Ritchie, 1986] Kernighan, B. W. and Ritchie, D. M. (1986). *C: A linguagem de programação*. Editora Campus-Elsevier. Tradução da edição original de *The C Programming Language*, Prentice-Hall, 1978.

[Levine, 2000] Levine, J. R. (2000). *Linkers and Loaders*. Morgan Kaufmann.

[Lu, 1997] Lu, H. (1997). Elf: From the programmer's perspective. NYNEX Science and Technology Notes.

[Motorola, 1989] Motorola, editor (1989). *M68000 8-/16-32-bit Microprocessors: User's Manual*. Prentice Hall.

[Paxson, 1995] Paxson, V. (1995). *Flex: A fast scanner generator*. Free Software Foundation.

[Skinner, 1988] Skinner, T. P. (1988). *Assembly Language Programming for the 68000 Family*. John Wiley and Sons.

[Stroustrup, 2001] Stroustrup, B. (2001). *A linguagem de programação C++*. Bookman, terceira edition. Tradução da terceira edição original, *The C++ Programming Language*, Addison-Wesley, 1997. Primeira edição em 1985.

[Ullman, 1976] Ullman, J. D. (1976). *Fundamental Concepts of Programming Systems*. Addison-Wesley.

[Wirth, 2005] Wirth, N. (2005). *Compiler Construction*. Addison-Wesley. Revisão do autor de livro publicado originalmente em 1996.

[Workman, 2007] Workman, D. A. (2007). Discrete structures: Course notes. `http://www.cs.ucf.edu/~workman/cot4210/`.

Índice Remissivo

Soluções dos exercícios

Capítulo 1

1.1 (a) 70 bits

(b) 17 bits

(c) 40 bits

1.2 Porque não haveria como diferenciar os dois campos de endereço para delimitar a origem e o destino. Por exemplo, as duas instruções do enunciado teriam a mesma codificação binária.

1.3 (a) 00000100

(b) 01000100

(c) 11101111

(d) 10001000

1.4 (a)
```
LOAD 0, R0
ADD R0, R2, R2
BZERO R2, 10
```

(b)
```
BZERO R3, 15
ADD R2, R2, R2
STORE R1, 9
LOAD 6, R3
```

1.5 (a) Mais um bit no código de operação.

(b) Mais um bit na referência para registrador.

(c) Sem impacto no código de operação.

(d) Mais quatro bits na referência à posição de memória.

Se todas as modificações forem implantadas, o novo formato da instrução será de 14 bits: 3 para o código de operação e 11 para os endereços de um registrador e de uma posição de memória; a instrução ADD terá dois bits não utilizados na sua codificação.

1.6

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char **argv) {
    ifstream arq;
    char ch;

    arq.open(argv[1]);
    while (! arq.eof()) {
        arq.get(ch);
        cout << ch;
    }
    arq.close();
}
```

1.7

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char **argv) {
    ifstream entrada;
    ofstream saida;
    char ch;

    entrada.open(argv[1]);
    saida.open(argv[2]);
    while (! entrada.eof()) {
        entrada.get(ch);
        saida.put(ch);
    }
    entrada.close();
    saida.close();
}
```

1.8

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char **argv) {
    ifstream entrada;
    char ch;
    int caracteres=0, palavras=0, linhas=0;
    bool novaPalavra=false;

    entrada.open(argv[1]);
    while (! entrada.eof()) {
        entrada.get(ch);
        ++caracteres;
    if (ch == '\n')
        ++linhas;
    if (ch == ' ' || ch == '\t' || ch == '\n')
        novaPalavra = false;
    else if (! novaPalavra) {
        ++palavras;
        novaPalavra = true;
        }
    }
    entrada.close();
    cout << caracteres << " " << palavras << " " << linhas << endl;
}
```

1.9 (a) O identificador é uma palavra reservada.

(b) A constante tem um dígito inválido para a base octal.

(c) O identificador tem um caractere inválido.

(d) O valor da constante extrapola a faixa representável pela variável.

1.10

```
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
    int total=0;
    while (argc-- > 1)
        total += atoi(argv[argc]);
    cout << total << endl;
}
```

1.11 (i) Marcação de final fora de ordem:

```
<autor>Machado de Assis <ano>1900 </autor></ano>
```

(ii) Caractere inválido para nome de elemento:

```
<#tombo>259</#tombo>
```

(iii) Ausência de elemento raiz:

```
<livro>Livro 1</livro>
<livro>Livro 2</livro>
```

1.12 Na ausência de um esquema de documento, o analisador não poderia saber se um elemento livro pode ou não conter outro elemento livro. O seguinte fragmento é sintaticamente correto, pelas regras de formação de XML:

```
<livro>The Complete Sherlock Holmes
  <livro>A study in scarlet</livro>
  ...
</livro>
```

Capítulo 2

2.1 (a) $C_1 = \{1, 2, 3, 4, 5, 6\}$

(b) $C_2 = \{4, 5, 6, 7, 8, 9\}$

(c) $C_3 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

(d) $C_4 = \{4, 5, 6\}$

(e) $C_5 = \{1, 2, 3\}$

(f) $C_6 = \{7, 8, 9\}$

(g) $C_7 = \emptyset$

(h) $C_8 = \{1, 2, 3, 7, 8, 9\}$

2.2 (a) Verdadeira.

(b) Falsa.

(c) Falsa.

(d) Verdadeira.

2.3 (a) $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

(b) $\{\emptyset\}, \{\{1\}, \{2\}, \{3\}\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}, \{\{1, 2, 3\}\}$

2.4 (a) 1, 010, 0011100

(b) 100, 110000, 111000000

(c) ε, 010, 010100

2.5 (a) $G_{2.5a} = (\{0,1\}, \{S, U, Z\}, P_{2.5a}, S)$; $P_{2.5a}$ é o conjunto com elementos:

$$\begin{aligned} S &\rightarrow ZUZ \\ Z &\rightarrow 0Z \\ Z &\rightarrow \varepsilon \\ U &\rightarrow 1U \\ U &\rightarrow 1 \end{aligned}$$

(b) $G_{2.5b} = (\{0,1\}, \{S\}, \{S \rightarrow 1S00, S \rightarrow 100\}, S)$

(c) $G_{2.5c} = (\{0,1\}, \{S\}, \{S \rightarrow 01S0, S \rightarrow \varepsilon\}, S)$

2.6 (a) $Z \Rightarrow 0Z1 \Rightarrow 00Z11 \Rightarrow 000Z111$

(b) $Z \Rightarrow 0Z1 \Rightarrow 01$

(c) $Z \Rightarrow 0Z1 \Rightarrow 00Z11 \Rightarrow 000Z111 \Rightarrow 0000Z1111 \Rightarrow 00001111$

2.7 (a) Sensível ao contexto (Tipo 1).

(b) a ; aa ; $aaaa$; $aaaaaaaa$

(c) Sentenças compostas por 2^n ocorrências do símbolo a, para $n \geq 0$.

2.8 $(\{a\}, \{S\}, \{S \rightarrow aaS, S \rightarrow \varepsilon\}, S)$

Exemplos: $\varepsilon, aa, aaaa, aaaaaa, aaaaaaaa$

2.9 $(\{a, b, c\}, \{S, X\}, \{S \rightarrow aX, X \rightarrow bX, X \rightarrow cX, X \rightarrow \varepsilon\}, S)$

Exemplos: $a, ab, ac, abc, acb, abccb$

2.10 $(\{a, b\}, \{S, A\}, \{S \rightarrow ba, S \rightarrow Ab, A \rightarrow Aa, A \rightarrow \varepsilon\}, S)$

Exemplos: $ba, b, ab, aab, aaab, aaaab$

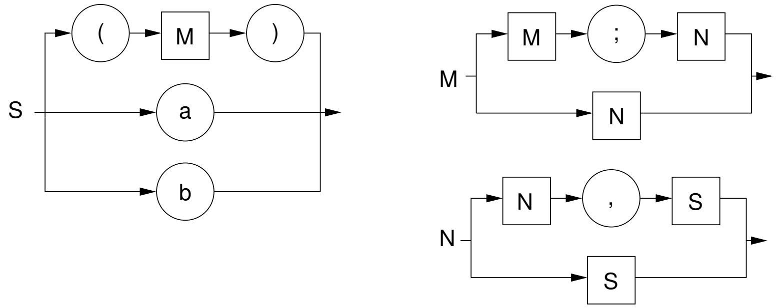
2.11 $(\{x, y, z\}, \{S, R, T, V\}, \{S \rightarrow RTV, R \rightarrow xR, R \rightarrow \varepsilon, T \rightarrow y, T \rightarrow z, V \rightarrow Vz, V \rightarrow \varepsilon\}, S)$

Exemplos: $y, z, xy, xz, xyz, xxzz$

2.12 (a) $(\{a, b, ;, \text{‘,’}, (,)\}, \{S, M, N\}, P_{2.12}, S)$, com $P_{2.12}$ contendo as seguintes produções:

$$\begin{aligned} S &\rightarrow (M) & M &\rightarrow M; N \\ S &\rightarrow a & M &\rightarrow N \\ S &\rightarrow b & N &\rightarrow N, S \\ & & N &\rightarrow S \end{aligned}$$

(b)



(c) $a \;:\; S \Rightarrow a$

$(b) \;:\; S \Rightarrow (M) \Rightarrow (N) \Rightarrow (S) \Rightarrow (b)$

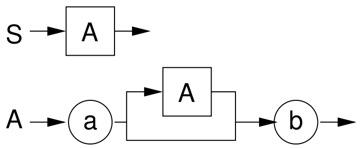
$(a;b,a) \;:\; S \Rightarrow (M) \Rightarrow (M;N) \Rightarrow (N;N) \Rightarrow (S;N) \Rightarrow (S;N,S)$
$\Rightarrow (S;S,S) \Rightarrow (a;S,S) \Rightarrow (a;b,S) \Rightarrow (a;b,a)$

2.13 (a) Gramática livre de contexto (Tipo 2).

(b)

```
<S> ::= <A>
<A> ::= a <A> b | a b
```

(c)

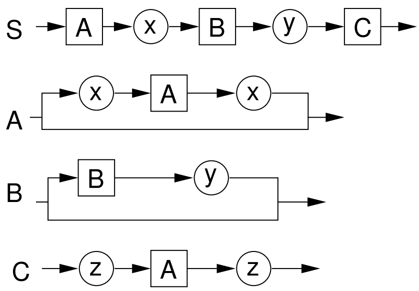


(d) $S \Rightarrow A \Rightarrow aAb \Rightarrow aabb$

2.14 (a)

```
<S> ::= <A> x <B> y <C>
<A> ::= x <A> x | ""
<B> ::= <B> y | ""
<C> ::= z <A> z
```

(b)

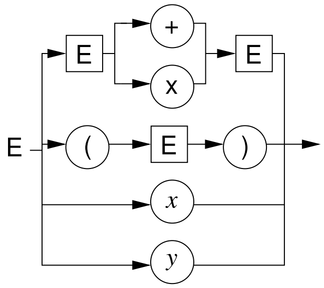


(c) $S \Rightarrow AxByC \Rightarrow xAxxByC \Rightarrow xxxByC \Rightarrow xxxByyC \Rightarrow xxxyyC \Rightarrow xxxyyzAz \Rightarrow xxxyyzxAxz \Rightarrow xxxyyzxxz$

2.15 (a) Gramática livre de contexto (tipo 2).

(b) <E> ::= <E> + <E> | <E> × <E> | (<E>) | x | y

(c)



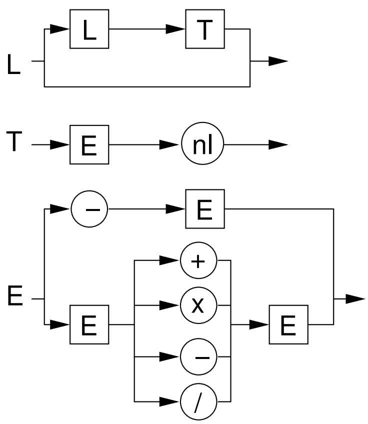
(d) i. $E \Rightarrow E + E \Rightarrow x + E \Rightarrow x + E \times E \Rightarrow x + x \times E \Rightarrow x + x \times y$

ii. $E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow x + E \times E \Rightarrow x + x \times E \Rightarrow x + x \times y$

2.16 (a) $(\{nl, int, -, +, *, /\}, \{L, T, E\}, P_{2.16}, L)$; $P_{2.16}$ é o conjunto com elementos:

$L \to LT$	$E \to E + E$
$L \to \varepsilon$	$E \to E - E$
$T \to E\ \ nl$	$E \to E * E$
$E \to int$	$E \to E/E$
$E \to -E$	

(b)



(c) i. `int nl`

ii. `int + int nl int * int nl`

2.17 (a) $G_f = (\{a, b, x, z\}, \{S, T, U\}, P_f, S)$; os elementos de P_f são as produções do enunciado.

(b) i. $S \Rightarrow TU \Rightarrow xU \Rightarrow xb$

ii. $S \Rightarrow aSz \Rightarrow aTUz \Rightarrow abTUz \Rightarrow abxUz \Rightarrow abxbz$

iii. $S \rightarrow TU \Rightarrow xU \Rightarrow xUx \Rightarrow xbx$

Capítulo 3

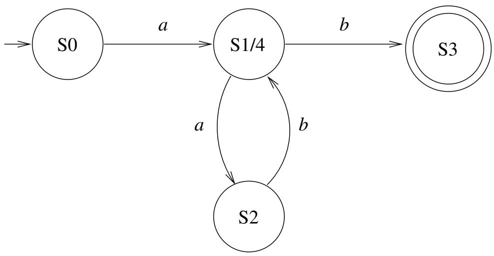
3.1

	s_0	s_1	s_2
a	s_1	—	—
b	s_2	—	—

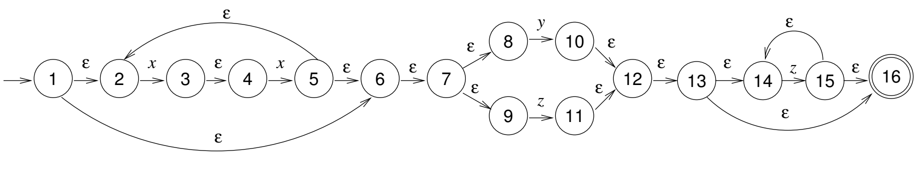
Estado inicial: s_0

Estados finais: s_1, s_2

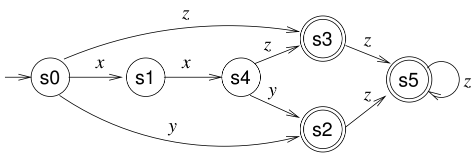
3.2



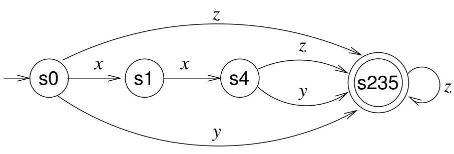
3.3 (a)



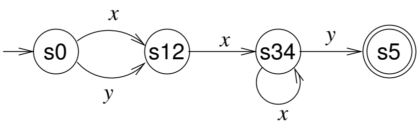
(b)



(c)



3.4 A expressão regular equivalente à gramática é `(x|y)xx*y`:



3.5 (a) `a(b|c)*`

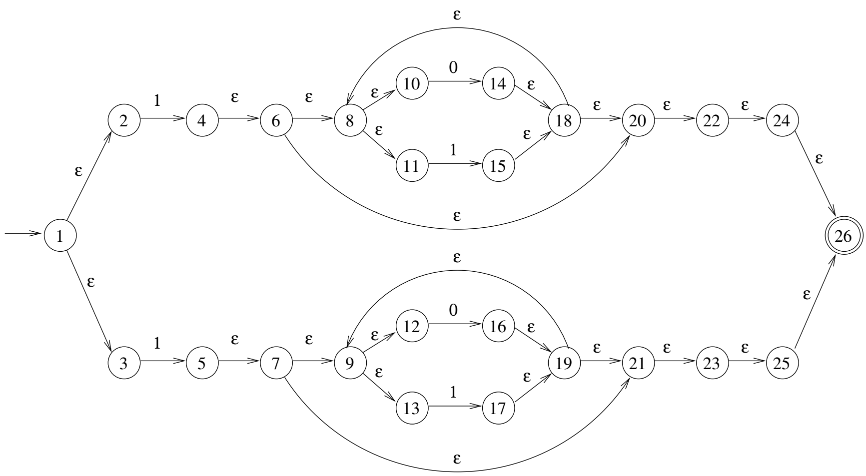
(b) `a(b|ca)*`

(c) `(ab*c)*`

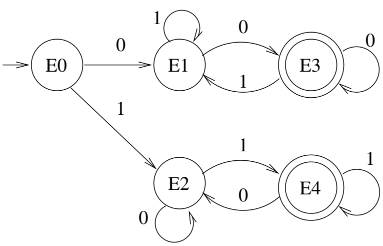
(d) `c*a(b|cc*a)*`

3.6 (a) `0(0|1)*0|1(0|1)*1`

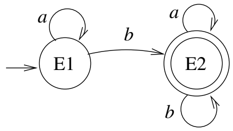
(b)



(c)



3.7 (a)



(b) `a*b(a|b)*`

3.8 (a) `(01+)*`

(b) `[+-]?([0-9]+,[0-9]*|[0-9]*,[0-9]+)([Ee][+-][0-9]+)?`

3.9 (a) Aceita (regra 1).

(b) Rejeitada.

(c) Rejeitada.

(d) Aceita (regra 1).

(e) Rejeitada.

(f) Aceita (regra 2).

(g) Rejeitada.

(h) Rejeitada.

(i) Aceita (regra 2).

(j) Rejeitada.

3.10 (a)

```
%{
#define D   300
#define O   301
#define H   302
#define F   303
#define B   304
#define X   399

double total = 0;
%}
%%
[1-9][0-9]*   { total += strtol(yytext,0,10); return D; }
0[0-7]*    { total += strtol(yytext,0,8); return O; }
0x[0-9A-Fa-f]+  { total += strtol(yytext,0,16); return H; }
[0-9]*\.[0-9]+|[0-9]+\.[0-9]* { total += strtod(yytext,0); return F; }
[ \n\t]+  { return B; }
.       { return B; }
<<EOF>> { return X; }
%%
int main(int argc, char *argv[]) {
  FILE *f_in;
  int tipoToken;
```

```
  int totalDec = 0,
      totalOct = 0,
      totalHex = 0,
      totalFlt = 0;

  if (argc == 2) {
    if (f_in = fopen(argv[1],"r"))
      yyin = f_in;
    else
      perror(argv[0]);
  }
  else
    yyin = stdin;

  while ((tipoToken = yylex()) != X)
    switch (tipoToken) {
      case D: ++totalDec;
              break;
      case O: ++totalOct;
              break;
      case H: ++totalHex;
              break;
      case F: ++totalFlt;
    }

  printf("Arquivo tem:\n");
  printf("\t %d valores decimais\n", totalDec);
  printf("\t %d valores octais\n", totalOct);
  printf("\t %d valores hexadecimais\n", totalHex);
  printf("\t %d valores reais\n", totalFlt);
  printf("Valor total: %f\n",total);
}
```

(b) Para valores em representação decimal e real:

```
...
-?[1-9][0-9]*   ...
...
-?[0-9]*\.[0-9]+|[0-9]+\.[0-9]* ...
...
```

3.11 (a) `[aeiouAEIOU][a-zA-Z]{8}[aeiouAEIOU]`

(b) `[a-zA-Z][a-zA-Z0-9]{4}`

(c) `[+-]?(0[1-7]*|0[xX][0-9a-fA-F]+|[1-9][0-9]*)`

3.12 (a) `a(a|n)*`

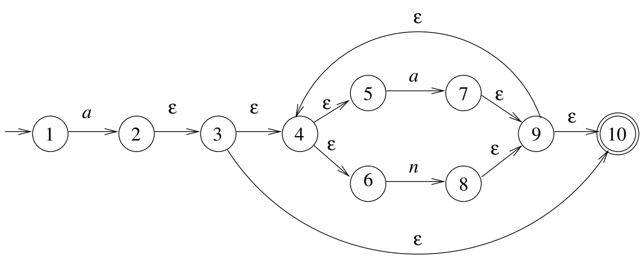
(b)

	e_1	e_2
a	e_2	e_2
b	—	e_2

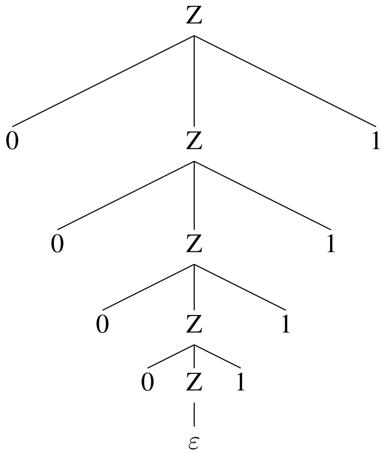
Inicial: e_1
Final: e_2

(c) `a`, `aa`, `an`, `aan`, `ana`, `aaa`, `ann`

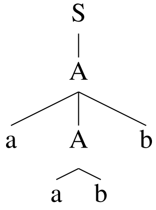
(d)



Capítulo 4

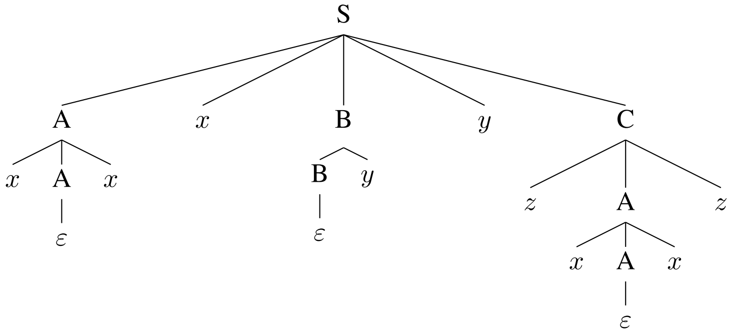
4.1 

4.2
- Árvore sintática:



- Derivação canônica (esquerda e direita): $S \Rightarrow A \Rightarrow aAb \Rightarrow aabb$
- Seqüência de reconhecimento mais à esquerda: 1, 2, 3
- Seqüência de reconhecimento mais à direita: 3, 2, 1

4.3 • Árvore sintática:



• Derivação canônica mais à esquerda:

$$S \Rightarrow AxByC \Rightarrow xAxxByC \Rightarrow xxxByC \Rightarrow xxxByyC \Rightarrow xxxyyC$$
$$\Rightarrow xxxyyzAz \Rightarrow xxxyyzxAxz \Rightarrow xxxyyzxxz$$

• Seqüência de reconhecimento mais à esquerda: 1, 2, 3, 4, 5, 6, 2, 3

• Derivação canônica mais à direita:

$$S \Rightarrow AxByC \Rightarrow AxByzAz \Rightarrow AxByzxAxz \Rightarrow AxByzxxz \Rightarrow AxByyzxxz$$
$$\Rightarrow Axyyzxxz \Rightarrow xAxxyyzxxz \Rightarrow xxxyyzxxz$$

• Seqüência de reconhecimento mais à direita: 3, 2, 5, 4, 3, 2, 6, 1

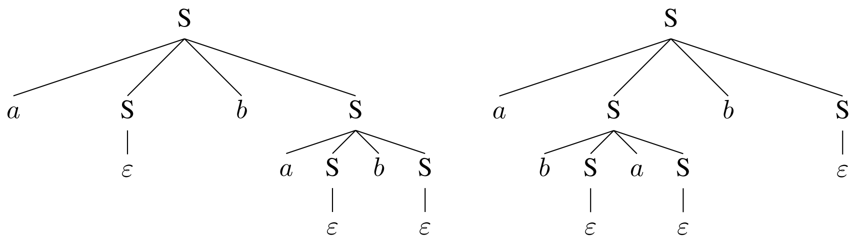
4.4 (a) i. $S \Rightarrow aSbS \Rightarrow aSbaSbS \Rightarrow aSbaSb \Rightarrow aSbab \Rightarrow abab$

ii. $S \Rightarrow aSbS \Rightarrow aSb \Rightarrow abSaSb \Rightarrow abSab \Rightarrow abab$

(b) i. $S \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$

ii. $S \Rightarrow aSbS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$

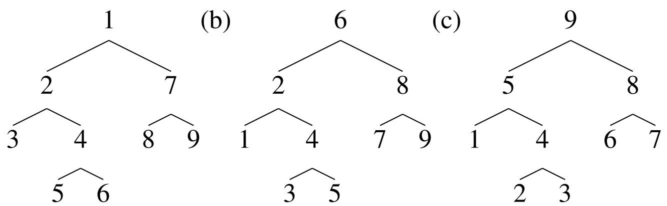
(c)



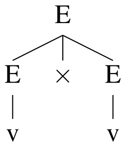
4.5 (a) a x b c y d v w z e f g

(b) b c x v w d y e f g z a

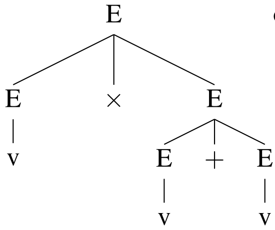
4.6 (a) (b) (c)



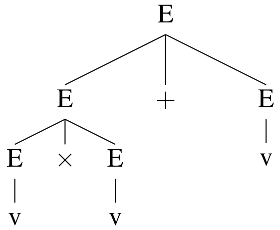
4.7 (a)



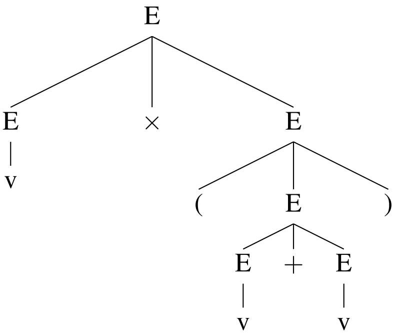
(b)



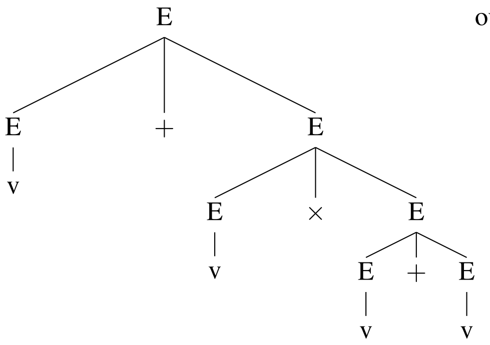
ou



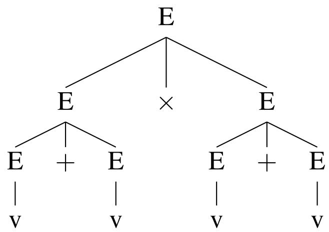
(c)



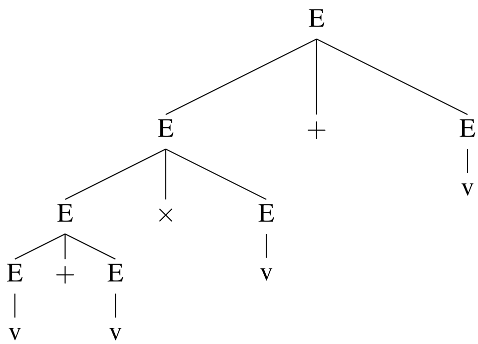
(d)



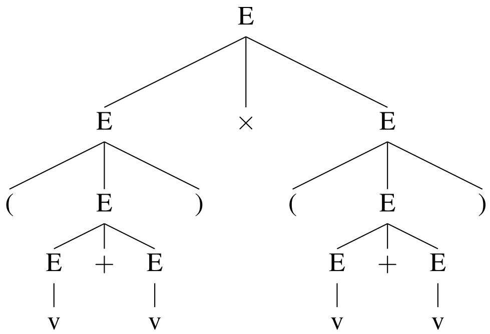
ou



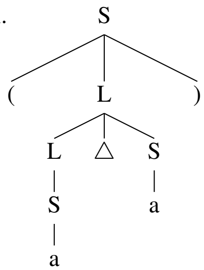
ou



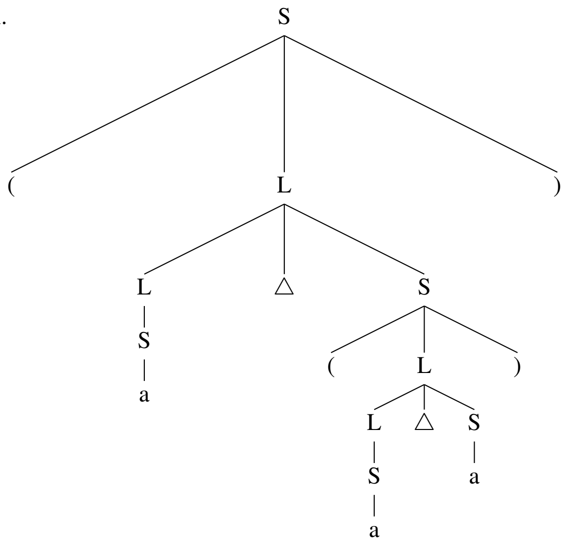
(e)



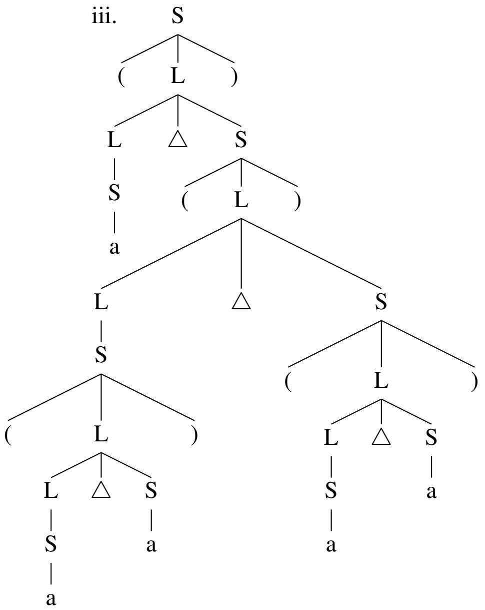
4.8 (a) i.



ii.



iii.



(b) Gramática com produções sem recursão à esquerda:

P1. $S \rightarrow (L)$
P2. $S \rightarrow a$
P3. $L \rightarrow SL'$
P4. $L' \rightarrow \triangle SL'$
P5. $L' \rightarrow \varepsilon$

Tabela sintática:

	(	)	$\triangle$	a	$
S	P1			P2	
L	P3			P3	
L'		P5	P4		

Reconhecimento das sentenças (X representa a eliminação dos tokens no topo da pilha e no início da sentença):

i. P1, X, P3, P2, X, P4, X, P2, X, P5, X.
ii. P1, X, P3, P2, X, P4, X, P1, X, P3, P2, X, P4, X, P2, X, P5, X, P5, X.
iii. P1, X, P3, P2, X, P4, X, P1, X, P3, P1, X, P3, P2, X, P4, X, P2, X, P5, X, P4, X, P1, X, P3, P2, X, P4, X, P2, X, P5, X, P5, X, P5, X.

(c)

	S	L	(	)	a	$\triangle$
S				$\succ$		$\succ$
L				$\approx$		$\approx$
(	$\prec$	$\prec\approx$	$\prec$		$\prec$	
)				$\succ$		$\succ$
a				$\succ$		$\succ$
$\triangle$	$\approx$		$\prec$		$\prec$	

(d)

	(	)	a	$\triangle$	$
S		R		R	
L		D		D	
(	D		D		
)		R		R	R
a		R		R	R
$\triangle$	D		D		
$	D		D		

Reconhecimento das sentenças (Ri indica redução pela produção i da gramática original):

i. D, D, R2, R4, D, D, R2, R3, D, R1.
ii. D, D, R2, R4, D, D, D, R2, R4, D, D, R2, R3, D, R1, R3, D, R1.
iii. D, D, R2, R4, D, D, D, D, R2, R4, D, D, R2, R3, D, R1, R4, D, D, D, R2, R4, D, D, R2, R3, D, R1, R3, D, R1, R3, D, R1.

4.9 (a) Gramática não é LL(1).

(b)

	S	A	a	b
S				
A				$\approx$
a		$\approx$	$\prec$	$\approx$
b				$\succ$

Há duas produções que terminam com b, mas como não há relação entre A (o símbolo que diferencia a terminação das duas strings do lado direito) e A (o lado esquerdo da outra produção), então a gramática é de precedência fraca.

(c)

	a	b	$
S			
A		D	R
a	D	D	
b		R	R
$	D		

(d) D, D, D, R3, D, R2, D, R? (sentença não reconhecida).

4.10 (a)

	a	e	o	l	x	$
S	P1		P1			
A	P2		P3			
B				P5	P4	

(b)

	a	e	o	l	x	$
S						
A				D	D	
B		D				
a				R	R	
e						R
o				R	R	
l		R				
x		R				
$	D		D			

(c) i. P1, P2, X, P4, X, X.

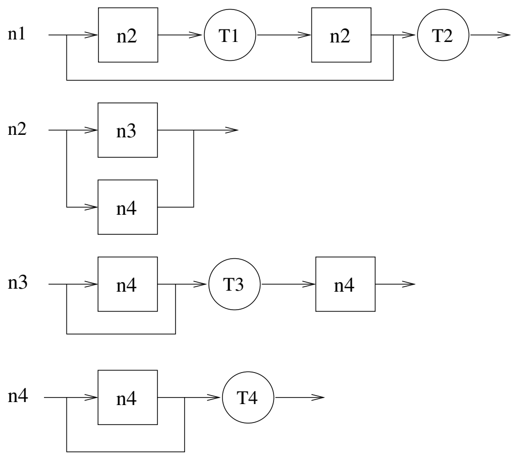
ii. D, R2, D, R4, D, R1.

4.11 (a) $G_{4.11} = (\{T_1, T_2, T_3, T_4\}, \{n_1, n_2, n_3, n_4\}, P_{4.11}, n_1)$; elementos de $P_{4.11}$:

$$n_1 \to n_2 T_1 n_2 T_2 \qquad n_3 \to n_4 T_3 n_4$$
$$n_1 \to T_2 \qquad n_3 \to T_3 n_4$$
$$n_2 \to n_3 \qquad n_4 \to n_4 T_4$$
$$n_2 \to n_4 \qquad n_4 \to T_4$$

(b)
```
<n1> ::= <n2> T1 <n2> T2 | T2
<n2> ::= <n3> | <n4>
<n3> ::= <n4> T3 <n4> | T3 <n4>
<n4> ::= <n4> T4 | T4
```

(c)



Capítulo 5

5.1 (a) O compilador incorpora ao nome da função o prefixo `_Z9` e um sufixo que é definido pelos tipos dos parâmetros da função; no exercício, os sufixos usados são `i` para cada parâmetro do tipo inteiro e `v` quando não há parâmetros (void). Outros possíveis sufixos são `b` (bool), `c` (char), `s` (short), `l` (long), `f` (float) e `d` (double).

(b) O tipo de retorno não influencia na decoração de nomes. Assim, a segunda e a terceira versão da função geram o mesmo nome decorado, produzindo um conflito na tabela de símbolos.

5.2 Apenas as operações (e) e (f) têm problemas, indicados pela verificação de tipos.

Capítulo 6

6.1 i. (a) Sem otimização:

```
_t1 := a + b
_t2 := _t1 + c
_t3 := b + c
_t4 := _t2 * _t3
  x := _t4 - d
```

(b) Propriedade algébrica e subexpressão comum:

```
_t1 := b + c
_t2 := a + _t1
_t3 := _t2 * _t1
  x := _t3 - d
```

(c)

1	+	b	b	_t1
2	+	a	_t1	_t2
3	*	_t2	_t1	_t3
4	-	_t3	d	x

(d)

1	+	b	c
2	+	a	(1)
3	*	(2)	(1)
4	-	(3)	d
5	:=	(4)	x

ii. (a) Sem otimização:

```
_t1 := a * b
param c
param 2
_t2 := call pow, 2
  r := _t1 * _t2
```

(b) Propriedade algébrica

```
_t1 := a * b
_t2 := c * c
  r := _t1 + _t2
```

(c)

1	*	a	b	_t1
2	*	c	c	_t2
3	+	_t1	_t2	r

(d)

1	*	a	b
2	*	c	c
3	+	(1)	(2)
4	:=	(3)	r

iii. (a) Sem otimização

```
_t1 := 4 * i
_t2 := i - 1
_t3 := 4 * _t2
_t4 := 4 * i
b[_t4] := a[_t1] + a[_t3]
```

(b) Uso de propriedade algébrica e eliminação de subexpressão comum

```
_t1 := 4 * i
_t2 := _t1 - 4
b [_t1] := a[_t1] + a[_t2]
```

(c)

1	*	4	i	_t1
2	-	_t1	4	_t2
3	+	a[_t1]	a[_t2]	b[_t1]

(d)

1	*	4	i
2	-	(1)	4
3	+	a[(1)]	a[(2)]
4	:=	(3)	b[(1)]

6.2 Sem considerar possíveis otimizações no código assembly:

i. **(a) 3 endereços:**

```
ADD  b, c, _t1
ADD  a, _t1, _t2
ADD  _t2, _t1, _t3
SUB  _t3, d, x
```

(b) 2 endereços:

```
MOVE b,  _t1
ADD  _t1, c
MOVE a, _t2
ADD  _t2, _t1
MOVE _t2, _t3
ADD  _t3, _t1
MOVE _t3, x
SUB  x, d
```

(c) 1 endereço:

```
LOAD  b
ADD   c
STORE _t1
LOAD  a
ADD   _t1
STORE _t2
LOAD  _t2
ADD   _t1
STORE _t3
LOAD  _t3
SUB   d
STORE x
```

(d) 0 endereços:

```
PUSH b
PUSH c
ADD
POP  _t1
PUSH a
PUSH _t1
ADD
POP  _t2
PUSH _t2
PUSH _t1
ADD
POP  _t3
PUSH _t3
PUSH d
SUB
POP  x
```

ii. **(a) 3 endereços:**

```
MUL  a, b, _t1
MUL  c, c, _t2
ADD  _t1, _t2, r
```

(b) 2 endereços:

```
MOVE a,  _t1
MUL  _t1, b
MOVE c, _t2
MUL  _t2, c
MOVE _t1, r
ADD  r, _t2
```

(c) 1 endereço:

```
LOAD  a
MUL   b
STORE _t1
LOAD  c
MUL   c
STORE _t2
LOAD  _t1
ADD   _t2
STORE r
```

(d) 0 endereços:

```
PUSH a
PUSH b
MUL
POP  _t1
PUSH c
PUSH c
MUL
POP  _t2
PUSH _t1
PUSH _t2
ADD
POP  r
```

iii. Notação adicional:

`#N`: endereçamento imediato;

`p(i)`: endereçamento indexado (conteúdo de `p` deslocado de `i` bytes).

iv. **(a) 3 endereços:**

```
MUL  #4, i, _t1
SUB  _t1, #4, _t2
ADD  a(_t1), a(_t2), b(_t1)
```

(b) 2 endereços:

```
MOVE #4,  _t1
MUL  i, #4
MOVE _t1, _t2
SUB  _t2, #4
MOVE a(_t1), b(_t1)
ADD  b(_t1), a(_t2)
```

(c) 1 endereço:

```
LOAD  #4
MUL   i
STORE _t1
LOAD  _t1
SUB   #4
STORE _t2
LOAD  a(_t1)
ADD   a(_t2)
STORE b(_t1)
```

(d) 0 endereços:

```
PUSH #4
PUSH i
MUL
POP  _t1
PUSH _t1
PUSH #4
SUB
POP  _t2
PUSH a(_t1)
PUSH a(_t2)
ADD
POP  b(_t1)
```

6.3 (a) `a b + c + b c + * d -`

(b) `a b a c b d / + * - -`

(c) `a b + a c d - e f - * - g + h / -`

6.4

```
      i := 0
_L1:  if i >= 10 goto _L2
      _t1 := 4 * i
      a[_t1] := 0
      i := i + 1
      goto _L1
_L2:  ...
```

6.5 (a)

```
      i := 0
_L1:  if i >= 100 goto _L2
      eps := 0.001
      _t1 := 4 * i
      if a[_t1] = 0 goto _L3
      _t2 := 4 * i
      _t3 := 4 * i
      _t4 := 4 * i
```

```
      c[_t4] := b[_t2] / a[_t3]
      goto _L4
_L3:  _t5 := 4 * i
      c[_t5] := eps
_L4:  i := i + 1
      goto _L1
_L2:  ...
```

(b) Eliminação de subexpressões comuns e movimentação de código

```
      i := 0
      eps := 0.001
_L1:  if i >= 100 goto _L2
      _t1 := 4 * i
      if a[_t1] = 0 goto _L3
      c[_t1] := b[_t1] / a[_t1]
      goto _L4
_L3:  c[_t1] := eps
_L4:  i := i + 1
      goto _L1
_L2:  ...
```

Capítulo 7

7.1

Símbolo	**Valor** (hexadecimal)
DATA	FA
FOO	FA
FSPIL	11A
LOOP	10A
PRGM	100
SPIL	100
XYZ	FC

7.2 O módulo gerado pelo montador de ligação direta deve incluir informações para ajustes de relocação e ligação, que no módulo gerado pelo montador absoluto são resolvidos pelo programador.

7.3 Ajuste de relocação é interno a um módulo, em função da sua posição inicial de carregamento. Ajuste de ligação ocorre entre módulos distintos.

7.4 A ligação dinâmica em tempo de carga verifica disponibilidade e define endereços de todos os módulos referenciados pelo módulo principal durante o carregamento, antes de dar início à execução. A ligação dinâmica em tempo de execução faz esse ajuste apenas quando o módulo é efetivamente referenciado.

7.5 Quando é necessário descobrir a qual variável um endereço de memória está associado — por exemplo, em uma atividade de depuração (*debugging*) de código.

7.6 O módulo de carregamento estático deve incorporar todo o código necessário à execução do programa. Portanto, é maior que o módulo de carregamento dinâmico, que contém apenas o módulo principal e referências aos demais módulos.

7.7 (a)

Símbolo	**Valor** (hexadecimal)
CALC	021A
MAIN	0200
RESULT	0218

(b)

Posição	**Conteúdo**
...	...
0202	0000
0204	0214
...	...
0208	0000
020A	021A
...	...
020E	0000
0210	0218
...	...